

设计有效的 数据库系统

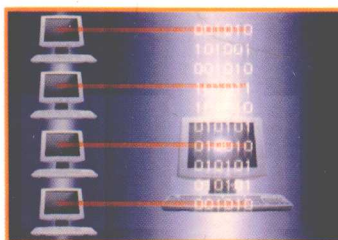
(美) Rebecca M. Riordan 著 何玉洁 张俊超 等译

"Riordan covers core skills for any developer—database design and development—in a perfect amount of detail. This book should be on every professional developer's reading list."

—Duncan Mackenzie, developer, Microsoft (MSDN)



Designing Effective Database Systems



REBECCA M. RIORDAN

Designing Effective Database Systems



机械工业出版社
China Machine Press

本书详细介绍关系数据库的设计原理，清晰地介绍了维度数据库建模，指导读者快速准确地掌握数据库设计的有效方法。书中指出了如何避免常见的设计结构隐患，这些隐患会使得数据库开发过程复杂化并降低可扩展性。本书作者是世界知名专家，已经帮助数以千计的专业人员掌握了数据库设计和开发技术。本书系统总结了作者多年开发数据库的经验，极具参考价值。

主要内容：

- 理解数据库模型、结构、关系和数据完整性原则
- 定义数据库系统目标、规则、范围和工作过程
- 构造精确的概念模型——关系、实体、域分析和规范化
- 构建有效、安全的数据库模式
- 掌握联机分析处理 (OLAP) 设计的元素——事实表、维度表、雪花架构及其他
- 组建和构造用于查询和报表的简单、有效的界面
- 学习基于Microsoft的Northwind样板数据库的实际例子

作者简介

Rebecca M. Riordan

已有15年以上的数据库设计、开发及应用经验。她获得了“微软最有价值程序员”的称号，并且经常在各种会议上发言，包括Microsoft TechEd。她还出版了多部著作，包括《*Seeing Data: Designing User Interfaces for Database Systems Using .NET*》(Addison-Wesley, 2005), 《*Designing Relational Database System*》(Microsoft Press, 1999), 《*Microsoft SQL Server 2000 Programming Step by Step*》(Microsoft Press, 2000), 以及《*ADO .NET Step by Step*》(Microsoft Press, 2002)。



www.PearsonEd.com



ISBN 7-111-18736-9



9 787111 187363



华章图书

上架指导：计算机/数据库

华章网站 <http://www.hzbook.com>

网上购书：www.china-pub.com

投稿热线：(010) 88379604

购书热线：(010) 68995259, 68995264

读者信箱：hzjsj@hzbook.com

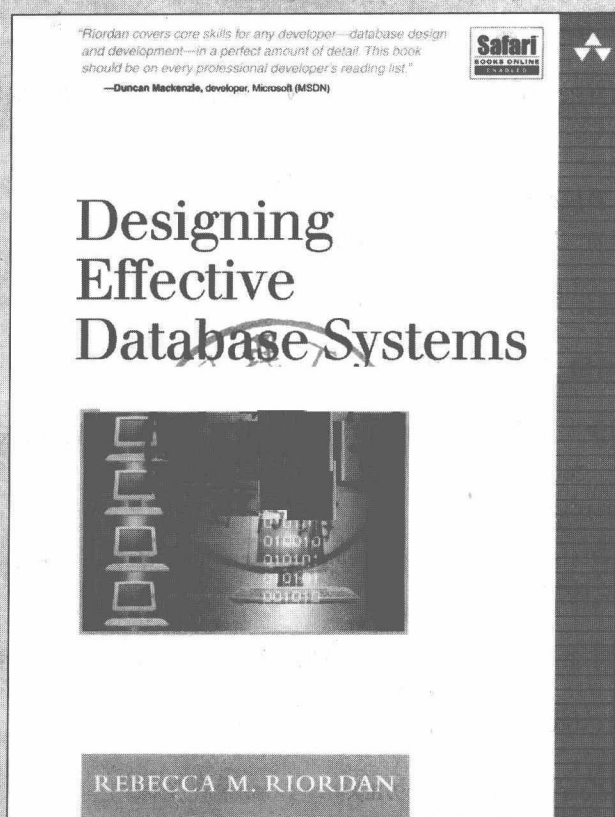
ISBN 7-111-18736-9/TP · 4724

定价：29.00 元

计 算 机 科 学 丛 书

设计有效的 数据库系统

(美) Rebecca M. Riordan 著 何玉洁 张俊超 等译



Designing Effective Database Systems



机械工业出版社
China Machine Press

本书系统地介绍了如何设计高效、高性能的数据库。首先详细阐明了关系设计的原理,清晰地介绍了维度数据库建模——从实用的角度来设计当今日益重要的分析型应用。接着分别阐明了传统数据库和用于数据仓库的维度数据库的分析和设计,指出了如何避免常见的结构隐患。本书广泛吸取了数据库设计方面的专家意见,可用性极强。本书适合软件开发人员、数据库设计人员参考。也可作为高等院校师生的参考书。

Simplified Chinese edition copyright © 2006 by Pearson Education Asia Limited and China Machine Press.

Original English language title: *Designing Effective Database Systems* (ISBN 0-321-29093-3) by Rebecca M. Riordan, Copyright © 2005.

All rights reserved.

Published by arrangement with the original publisher, Pearson Education, Inc., publishing as Addison Wesley.

本书封面贴有Pearson Education (培生教育出版集团) 激光防伪标签,无标签者不得销售。

版权所有,侵权必究。

本书法律顾问 北京市展达律师事务所

本书版权登记号: 图字: 01-2005-3617

图书在版编目(CIP)数据

设计有效的数据库系统 / (美) 里尔丹 (Riordan, R. M.) 著; 何玉洁等译. - 北京: 机械工业出版社, 2006.5

(计算机科学丛书)

书名原文: *Designing Effective Database Systems*

ISBN 7-111-18736-9

I. 设… II. ① 里… ② 何… III. 数据库系统 - 系统设计 IV. TP311.13

中国版本图书馆CIP数据核字 (2006) 第024466号

机械工业出版社 (北京市西城区百万庄大街22号 邮政编码 100037)

责任编辑: 罗媛媛 刘立卿

北京诚信伟业印刷有限公司印刷 · 新华书店北京发行所发行

2006年5月第1版第1次印刷

787mm × 1092mm 1/16 · 14印张

定价: 29.00元

凡购本书,如有倒页、脱页、缺页,由本社发行部调换
本社购书热线 (010) 68326294

出版者的话

文艺复兴以降，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的传统，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅筹划了研究的范畴，还揭橥了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短、从业人员较少的现状下，美国等发达国家在其计算机科学发展的几十年间积淀的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章图文信息有限公司较早意识到“出版要为教育服务”。自1998年开始，华章公司就将工作重点放在了遴选、移译国外优秀教材上。经过几年的不懈努力，我们与Prentice Hall, Addison-Wesley, McGraw-Hill, Morgan Kaufmann等世界著名出版公司建立了良好的合作关系，从它们现有的数百种教材中甄选出Tanenbaum, Stroustrup, Kernighan, Jim Gray等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及度藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力襄助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专程为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍，为进一步推广与发展打下了坚实的基础。

随着学科建设的初步完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都步入一个新的阶段。为此，华章公司将加大引进教材的力度，在“华章教育”的总规划之下出版三个系列的计算机教材：除“计算机科学丛书”之外，对影印版的教材，则单独开辟出“经典原版书库”；同时，引进全美通行的教学辅导书“Schaum's Outlines”系列组成“全美经典学习指导系列”。为了保证这三套丛书的权威性，同时也为了更好地为学校和老师服务，华章公司聘请了中国科学院、北京大学、清华大学、国防科技大学、复旦大学、上海交通大学、南京大学、浙江大学、中国科技大学、哈尔滨工业大学、西安交通大学、中国人民大学、北京航空航天大学、北京邮电大学、中山大学、解放军理工大学、郑州大学、湖北工学院、中国国家信息安全测评认证中心等国内重点大学和科研机构在计算机的各个领域的著名学者组成“专家指导委员会”，为我们提供选题意见和出版监督。

这三套丛书是响应教育部提出的使用外版教材的号召，为国内高校的计算机及相关专业

的教学度身订造的。其中许多教材均已为M. I. T., Stanford, U.C. Berkeley, C. M. U. 等世界名牌大学所采用。不仅涵盖了程序设计、数据结构、操作系统、计算机体系结构、数据库、编译原理、软件工程、图形学、通信与网络、离散数学等国内大学计算机专业普遍开设的核心课程,而且各具特色——有的出自语言设计者之手、有的历经三十年而不衰、有的已被全世界的几百所高校采用。在这些圆熟通博的名师大作的指引之下,读者必将在计算机科学的宫殿中由登堂而入室。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑,这些因素使我们的图书有了质量的保证,但我们的目标是尽善尽美,而反馈的意见正是我们达到这一终极目标的重要帮助。教材的出版只是我们的后续服务的起点。华章公司欢迎老师和读者对我们的工作提出建议或给予指正,我们的联系方法如下:

电子邮件: hzjsj@hzbook.com

联系电话: (010) 68995264

联系地址: 北京市西城区百万庄南街1号

邮政编码: 100037

专家指导委员会

(按姓氏笔画顺序)

尤晋元
石教英
张立昂
邵维忠
周克定
郑国梁
高传善
裘宗燕

王 珊
吕 建
李伟琴
陆丽娜
周傲英
施伯乐
梅 宏
戴 葵

冯博琴
孙玉芳
李师贤
陆鑫达
孟小峰
钟玉琢
程 旭

史忠植
吴世忠
李建中
陈向群
岳丽华
唐世渭
程时端

史美林
吴时霖
杨冬青
周伯生
范 明
袁崇义
谢希仁

对本书的赞誉

“设计数据库不是件简单的事情，是软件开发的基础部分，作者清晰地阐述了基本概念并介绍了其丰富的经验。”

——Patrick Birch，数据库和技术写作顾问

“这本书十分专业地引导读者进行数据库开发，并且在第一次开发时就能做对！”

——Kenneth D. Snell博士，Access开发者，微软Access最有价值程序员（MVP）

“本书的独特之处在于讲述关系数据库的设计方法既正统又通俗，并在此基础上有所深化！尤其是数据仓库的开发设计部分，很多数据库设计者都会受益匪浅。如果你正在寻觅一个设计事务数据库的框架，或者试图涉足分析数据库的领域，那么这本书都能给予你很大的帮助。”

——Paul Irvine，Via Training公司的副总裁，工程师

“如果你只想买一本关于数据库设计的书，那么就买这本。作者十分擅长用浅显的语言来解释复杂的技术问题。”

——Brendan Reynolds，Dataset IT系统开发人员，微软Access最有价值程序员

“Riordan选择了一个复杂的主题，但却使其简单化了。如果你对数据库设计项目一筹莫展的话，这本书一定能帮你摆脱困扰！”

——Mike Gunderloy，Application Development Trends资深编辑

“本书以通俗的语言和良好的结构覆盖了数据库设计和数据建模主题的广泛内容。”

——Amy Sticksel，Sticksel Data Systems, Inc

“在这本书中，Riordan的风格、才智和对细节的关注都是值得称道的。”

——Sandra Daigle，微软Access最有价值程序员

“这不仅仅是一本关于数据库理论的书，从开发一个数据库应用程序的洽谈阶段到完成阶段，作者都给予了我们十分实际的指导。”

——Dirk Goldgar，DataGnostics总裁，微软Access最有价值程序员

“这本书适合任何数据库设计和开发者，它囊括了各种核心技巧，包括规范化之类的基础技巧，以及如何构建系统以便生成报表和进一步扩展。这本书应当列为每位专业开发者的必读书目。”

——Duncan Mackenzie，微软MSDN开发人员

译者序

初次拿到本书原版时，原著的书名不禁让人有些许失望——又是一本数据库设计的书。随着数据库技术的迅速发展和广泛应用，市面上关于数据库设计的书籍比比皆是，其中的确不乏滥竽充数者。但是，当译者翻看完目录之后，就立刻被其吸引了。通读全书，其缜密的结构、丰富的内涵以及极具价值的指导意义确实让人无法释手。

与其他一些介绍数据库设计的同类书籍相比，本书具有以下独到之处：

第一，内容更丰富。这一点主要体现在两个方面。首先，作者打破了数据库设计仅仅针对面向事务的传统数据库的框架，而以新颖的角度引入了维度数据库的理论。这是数据仓库领域的基础，也是目前较前沿的面向分析的数据库应用技术。另外，在本书的最后部分，作者用较重的笔墨详尽阐述了设计用户界面的相关技术，不再局限于数据库的后台构建和设计。因为用户眼中的数据库系统就是一个“界面”，其成败直接关系到数据库系统的实用性。因此，这使得本书的实际参考价值更高。

第二，专家指导意义更强。由于本书作者自身具有多年的数据库设计和项目开发经验，并且她还是微软的最有价值程序员（MVP），因此，书中随处可见其融入的指导性意见。特别是本书的第三和第四部分，作者以其丰富的专家经验，指明了在设计数据库的各个环节中容易出现的问题以及应对的方法。此外，作者经常针对一个问题，提出多种实际解决方案供读者参考选用。

第三，语言亲切生动。与其他众多专业书籍不同的是，本书的语言读起来让人倍感亲切，大有与读者对话之感。作者在书中多处运用丰富的修辞表现手法，让本来生硬的理论和技术变得生动活泼，当然这也给译者造成了不小的翻译难度。译者在阅读过程中也不时为作者的幽默和智慧所叹服，这也是本书让人难以释手的主要原因之一。

本书适合有一定数据库理论基础，又非常希望能将理论知识应用到实际使用过程中的读者。此外，本书可同时作为学习数据库理论知识和实用开发技术的书籍，也可作为数据库设计和开发人员的数据库设计参考书。

本书由何玉洁负责组织翻译和审校，主要工作由何玉洁和张俊超完成，张帆、侯永胜、张俊、张丹丹、潘晓洁、黄婷儿、卢立能、李迎、崔晗等参加了本书部分章节的翻译。其中：张帆和侯永胜参加了第1章的翻译，张丹丹和潘晓洁参加了第2章的翻译，卢立能和黄婷儿参加了第3章的翻译，张俊参加了第6章的翻译，崔晗和李迎参加了第16章的翻译，张俊超负责第4、5、7章的翻译，其他章由何玉洁负责翻译。全书由何玉洁负责审校。是大家的认真工作使本书得以顺利完成，在此表示衷心的感谢。

限于译者水平，译校过程中难免有疏漏和不妥之处，恳请读者批评指正。

何玉洁
2005年11月

前 言

关系数据库的确是一个繁杂的问题，相比之下，其他类型的商业软件理解起来要容易很多。文字处理软件实际上就是高技术的打字机，并且显然退格键要比修正液好用得多。电子表格软件提供了一个大家十分熟悉的模板，甚至不需要会计了；而电子邮件从易于理解的角度来看与一个邮政系统十分类似。

但数据库与众不同。其他类型的软件都有一个现实世界的相似物。有时候，比如在Windows桌面系统中，这种相似物有一点空洞，但是它们都是十分亲近的，都触手可及。但是关系数据库则完全是虚构的。这就像几何学：虽然它们一般并不存在于现实世界中，但可以用来构建现实世界的模型。你最近一次为你和爱人倒上红酒，然后走到前廊上观赏湖面上“几何物的欢聚”是几时？

现在，我们在这里谈论的是数据库，不是表格。现实世界中存在很多表格：从电话簿到字典。而关系数据库则不然。你也不可能在“湖面上的欢聚”中找到它们。图书馆里的卡片文档与数据库有些接近，它们包含作者、书名和主题文件，但是它们仍旧把数据的集合分开了，只有该馆的图书管理员才能将它们关联起来。

这是一本关于设计数据库系统的书。初衷是想教读者一些如何将一个凌乱复杂的现实世界转变为有效的数据库设计的有关知识。我假定读者已经有一些开发经验，并且知晓有关计算机的常识，但读者可以没有数据库的知识背景。

在阅读完本书之后，你仍旧没法观赏数据库“在湖面上欢聚”，但是如果我实现了我的写作初衷的话，那么读者将能够设计和实现一个有关鱼、海鸥以及浮游生物对二者的影响的关系模型。

本书分为四部分。第一部分“关系数据库理论”包括关系模型的基本原理。这些的确是很枯燥的理论知识，但是不要担心，它们理解起来也很容易。第二部分“维度数据库理论”包含维度数据库的理论知识，维度数据库是一种用于分析的特殊类型的关系数据库。第三部分“设计数据库系统”介绍了分析和设计过程——如何将现实世界设计成一个可靠的数据库系统。最后，第四部分“设计用户界面”从用户的角度讨论了数据库系统中最重要的一个方面：用户界面。

虽然书中谈论的都是有关实现的问题，但这并不是一本“如何编程”的书。书中有一些编码示例，但是我已经将它们减小到最少，即便读者以前从来没学过任何编程语言，也应当能够读懂它们。数据库的例子基于Microsoft Access的Northwind样板数据库（SQL Server中的Northwind版本也十分类似）。当你读完本书的时候，就能获得开始构建数据库系统的大部分知识，并且可以借助参考文献中的资料来学习更好的编程风格。同时你将十分坚信你的数据构架十分完善，不会在以后的项目中给你带来麻烦。

目 录

出版者的话	
专家指导委员会	
对本书的赞誉	
译者序	
前言	

第一部分 关系数据库理论

第1章 基本概念	1
1.1 什么是数据库	1
1.2 数据库工具	3
1.2.1 数据库引擎	3
1.2.2 数据访问对象模型	4
1.2.3 数据定义环境	5
1.2.4 前端开发	5
1.3 关系模型	5
1.4 关系术语	6
1.5 数据模型	7
1.5.1 实体	8
1.5.2 属性	8
1.5.3 域	11
1.5.4 联系	12
1.5.5 实体联系图	13
1.6 小结	14
第2章 数据库结构	15
2.1 消除冗余	15
2.2 保证灵活性	17
2.3 基本原则	19
2.3.1 无损分解	19
2.3.2 候选码和主码	20
2.3.3 函数依赖	21
2.4 第一范式	22
2.5 第二范式	23
2.6 第三范式	25
2.7 进一步的规范化	26

2.7.1 Boyce/Codd范式	26
2.7.2 第四范式	27
2.7.3 第五范式	28
2.8 小结	29
第3章 联系	30
3.1 术语	30
3.2 联系建模	31
3.3 一对一联系	33
3.4 一对多联系	36
3.5 多对多联系	36
3.6 一元联系	37
3.7 三元联系	37
3.8 已知基数的联系	39
3.9 小结	40
第4章 数据完整性	41
4.1 完整性约束	41
4.1.1 域完整性	41
4.1.2 转换完整性	43
4.1.3 实体完整性	43
4.1.4 参照完整性	44
4.1.5 数据库完整性	45
4.1.6 事务完整性	45
4.2 实现数据完整性	46
4.2.1 未知值和不存在的值	46
4.2.2 冲突响应	48
4.2.3 声明的和过程的完整性	48
4.2.4 域完整性	48
4.2.5 实体完整性	49
4.2.6 参照完整性	52
4.2.7 其他类型的完整性	52
4.3 小结	53
第5章 关系代数	54
5.1 Null值和三值逻辑	55
5.2 关系运算	56

5.2.1 选择	56	9.1 生命周期模型	87
5.2.2 投影	56	9.2 数据库设计过程	90
5.2.3 连接	57	9.2.1 定义系统参数	90
5.2.4 除	60	9.2.2 定义工作过程	90
5.3 集合运算符	60	9.2.3 构建概念数据模型	90
5.3.1 并	60	9.2.4 准备数据库模式	90
5.3.2 交	61	9.2.5 设计用户界面	91
5.3.3 差	61	9.3 关于设计方法和标准的提示	91
5.3.4 笛卡儿积	63	第10章 定义系统参数	92
5.4 特殊的关系运算符	63	10.1 定义系统目标	92
5.4.1 总结	63	10.2 开发设计标准	95
5.4.2 扩展	64	10.2.1 直接衡量标准	96
5.4.3 重命名	64	10.2.2 环境标准	96
5.4.4 变换	64	10.2.3 一般设计策略	97
5.4.5 上卷	65	10.3 定义系统范围	98
5.4.6 立方体	66	10.4 小结	100
5.5 小结	66	第11章 定义工作过程	101
第二部分 维度数据库理论			
第6章 维度的基本概念	67	11.1 确定当前工作过程	101
6.1 维度数据库模型	67	11.1.1 与用户交流	101
6.2 术语	70	11.1.2 确定任务	102
6.3 商务智能的浓缩历史	71	11.2 分析工作过程	105
6.4 小结	72	11.3 将工作过程文档化	106
第7章 事实表	73	11.4 用户情景	107
7.1 事实表的结构	73	11.5 小结	108
7.2 事实属性的特征	74	第12章 概念数据模型	109
7.2.1 粒度	75	12.1 确定数据对象	109
7.2.2 事实表的类型	76	12.2 定义联系	112
7.2.3 异类事实	77	12.2.1 联系的基数	113
7.3 小结	79	12.2.2 联系的可选性	114
第8章 维度表	80	12.2.3 联系的属性	114
8.1 维度表的结构	80	12.2.4 联系的附加约束	114
8.2 雪花化	83	12.3 复查实体	114
8.3 改变维度	84	12.3.1 实体和问题域之间的联系	115
8.4 小结	86	12.3.2 影响实体的工作过程	115
第三部分 设计数据库系统			
第9章 设计过程	87	12.3.3 实体间的交互	115
		12.3.4 业务规则和约束	116
		12.3.5 属性	116
		12.4 域分析	117
		12.5 限制值的范围	118

12.6	规范化	119
12.7	小结	119
第13章	数据库模式	120
13.1	系统架构	120
13.1.1	编码架构	120
13.1.2	数据架构	124
13.2	数据库模式组件	130
13.2.1	定义表和联系	130
13.2.2	视图和查询	131
13.3	安全性	132
13.4	小结	134
第14章	交流设计	135
14.1	读者和目标	135
14.2	文档结构	135
14.3	执行小结	136
14.4	系统概貌	137
14.5	工作过程	137
14.6	概念数据模型	138
14.7	数据库模式	139
14.8	用户界面	139
14.8.1	界面原型法	140
14.8.2	界面说明书	140
14.9	修订管理	141
14.10	小结	141

第四部分 设计用户界面

第15章	作为中间媒介的用户界面	143
15.1	有效的界面	143
15.2	界面模型	144
15.3	用户层次	145
15.3.1	初学者	145
15.3.2	中级用户	145
15.3.3	专业用户	145
15.4	让用户管理	146
15.5	减轻记忆的负担	147
15.6	保持一致性	148
15.7	小结	150
第16章	用户界面架构	151
16.1	支持工作过程	151

16.2	文档架构	152
16.2.1	单文档界面	152
16.2.2	多文档界面	154
16.3	小结	159
第17章	在窗体设计中描述实体	160
17.1	简单实体	160
17.2	一对一联系	162
17.3	一对多联系	162
17.4	层次	165
17.5	多对多联系	166
17.6	小结	168
第18章	选择Windows控件	169
18.1	表达逻辑数据	170
18.2	表达多个值的集合	170
18.2.1	从一组值中获取单个值	171
18.2.2	获取一组值	172
18.3	表达数字和日期	173
18.4	表达文本数据	175
18.5	小结	176
第19章	维护数据库的完整性	177
19.1	完整性约束的类别	177
19.2	内在约束	178
19.2.1	数据类型	178
19.2.2	格式	179
19.2.3	长度	179
19.2.4	空值	179
19.2.5	范围	180
19.2.6	实体和参照完整性约束	180
19.3	业务约束	182
19.3.1	偶然输入	183
19.3.2	现实与系统模型的对比	183
19.4	小结	185
第20章	报表	186
20.1	排序、检索和过滤数据	186
20.1.1	排序数据	187
20.1.2	通过选择过滤	187
20.1.3	通过窗体过滤	187
20.1.4	高级过滤和排序	188
20.1.5	微软自然语言查询	189

20.2 生成标准报表	189	21.2 被动帮助机制	199
20.2.1 清单报表和明细报表	189	21.2.1 帮助记忆的访问键	199
20.2.2 总结报表	190	21.2.2 工具提示	200
20.2.3 基于窗体的报表	190	21.2.3 状态栏	201
20.2.4 报告界面	190	21.3 反应帮助机制	202
20.2.5 处理打印机错误	191	21.3.1 联机帮助	202
20.2.6 自动和随选打印	192	21.3.2 “What’s This?” 提示	203
20.3 生成特殊报表	193	21.3.3 可听见的反馈	204
20.3.1 报表设计器	193	21.3.4 错误消息	205
20.3.2 自定义的报表设计	193	21.4 主动帮助机制	206
20.3.3 标准信件	196	21.5 用户培训	206
20.4 小结	197	21.6 小结	207
第21章 用户帮助	198	术语表	208
21.1 用户级别	198	参考文献	212

第一部分 关系数据库理论

第1章 基本概念

关系数据库这个神秘的概念到底是什么含意呢？简单地说，数据库是一种高效且有力的存储和操作信息的工具。“高效和有力”有三个方面的含义：其一，它意味着数据是被保护的，免于意外的丢失或损坏；其二，它不需要占用过多的资源（包括人和计算机）；其三，它可以在可接受的性能约束下，以恰当的方式检索数据。要符合关系型的要求，数据库必须实现关系模型。关系模型是根据一系列规则来描述现实世界某些方面的一种方法，这些规则是E.F.Codd博士在20世纪60年代末首次提出来的。

在理论上，一个关系数据库应该是从草图开始构建，但实际上，我们通常使用数据库管理系统（DBMS）的服务。一个DBMS有时也被称为是关系型数据库管理系统（RDBMS），但技术上，DBMS必须符合大约300条规则才能被称为是关系型的，并且就我所知，到目前为止还没有一个商业化的系统完全满足这些规则。本书将介绍两个数据库管理系统：Microsoft Jet和Microsoft SQL Server。

关系数据库是关系模型（数据模型）的物理实现，清楚地区分这两个概念是很重要的。正如我们在第二部分将要看到的，维度模型也可以在关系型DBMS中实现。如果将概念层和物理层相混淆的话，你将会陷入到一个十分混乱的状态（当然，这也是经验证明的）。

1.1 什么是数据库

数据库这个术语几乎和“面向对象程序设计”一样的耳熟能详。单词“数据库”可用于描述任何事情，从单一的数据集合，比如电话号码列表，到复杂的工具集合，比如SQL Server，又或者是介于简单和复杂之间的一些事物。这种缺乏准确性的描述并不一定是坏事，毕竟这只是一个自然语言，但这种不准确性对我们的目标十分不利，因此，我在这里将尽量给出数据库更准确的定义。图1-1说明了这里所讨论的术语之间的关系。我们将在本章定义这些术语，并在本书的其他部分详细讨论它们。

尽管关系型数据库在现实世界中没有对应的概念，但大多数关系型数据库都是为了模型化现实世界的某些方面，我们将这部分现实世界称之为**问题域**（problem space）。由于问题域的自然特性决定了它是混乱和复杂的——如果不是这样，我们也就不需要构建它的模型了。将你所设计的数据库系统限制在一个具体的、定义准确的对象集合及其交互中是项目成功与否的关键，只有这样才能对系统的范围作出明智的决策。

术语**数据模型**（data model）代表问题域的概念描述，使用关系模型时，这个概念包括实体、实体的属性（例如，一个客户是一个实体，他可能具有姓名、地址等属性）以及实体约束的定义（例如，客户名字段不允许空值）。使用维度模型时，这个概念包含事实和维度，我们将在第二部分讨论维度模型问题。

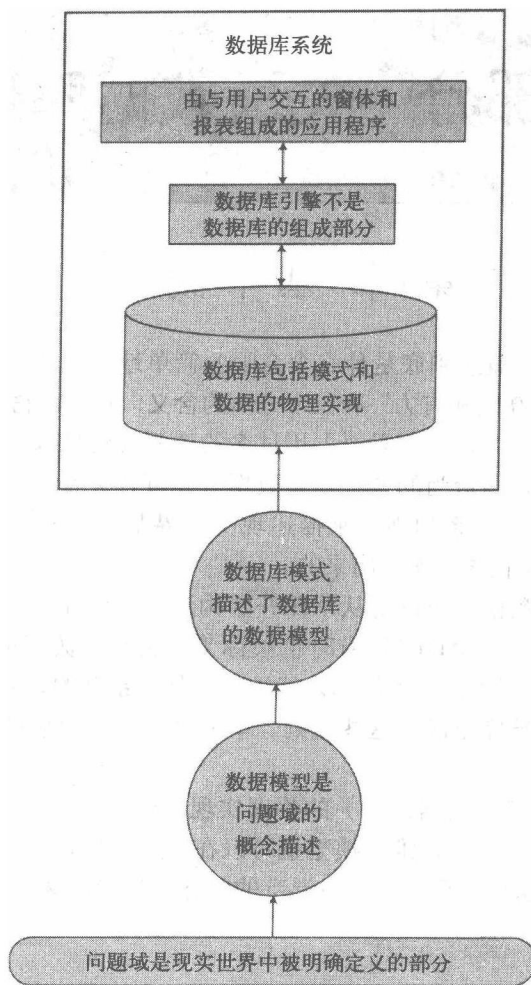


图1-1 关系型数据库术语

数据模型也包含对实体间的联系以及这些联系上的约束的描述。例如，经理不允许有超过5位直接向他汇报的个人。数据模型不包含任何对系统物理层的引用。

物理层的定义——表和视图的实现——称为**数据库模式**（database schema）或简称为**模式**（schema），它是将概念模型转换为DBMS可以实现的物理表达。注意模式仍然是概念上的而不是物理上的。模式只是数据模型按数据库引擎方式描述的另一种表达——例如表、触发器等等。使用数据库引擎的一个好处是用户永远不需要处理物理实现；你可以完全忽略B-树、叶结点以及其他底层物理概念。

一旦向数据库引擎定义好了你希望的数据形式，就可以使用任何代码或交互环境（比如Microsoft Access、SQL Server Enterprise Manager），数据库引擎将创建一些物理对象（通常是在硬盘的某些地方创建这些对象，但并不一定都是这样），然后用户就可以在这些对象中存储数据了。结构和数据的结合就是我们所说的**数据库**。这个数据库包括存储数据的物理表、定义的视图、查询，用于以不同方式检索数据的存储过程，以及将用于加强数据保护的规则。

术语“数据库”不包括应用程序，应用程序是由与用户交互的窗体、报表组成的，同时，

数据库也不包括类似中间件、Internet Information Server (Internet 信息服务, 用于前端和后端的连接) 的内容, 术语“数据库”也不包含数据库引擎。因此, Access 的.mdb文件是一个数据库, 而Microsoft Jet是一个数据库引擎。实际上, 一个.mdb文件可以包含应用程序对象, 比如, 窗体、报表, 这是我们后面将要讨论的内容。

为了包含所有这些内容——应用程序、数据库、数据库引擎以及中间件, 我们将使用**数据库系统** (database system) 这个术语。所有这些构成一个生产系统的软件和数据都属于数据库系统的一部分。

1.2 数据库工具

尽管本书的重点是设计而不是实现, 因为抽象理论并没有太大的价值, 除非你已经知道如何应用这些理论, 因此, 在这本书中, 我们将介绍很多使用Microsoft提供的工具构建关系数据库的方法, 这样的工具有很多, 并且Microsoft引入的新方法和工具也是层出不穷, 因此, 现在让我们花一点时间来看一下这些工具都是什么, 以及它们是如何协同工作的。图1-2显示了我们讨论的工具, 从设计者的角度来看, 可以很简单地将这些工具看成是将系统从抽象模型转换为一个现实的生产系统所需的工具, 并且这些工具之间的组合关系如图1-2所示。

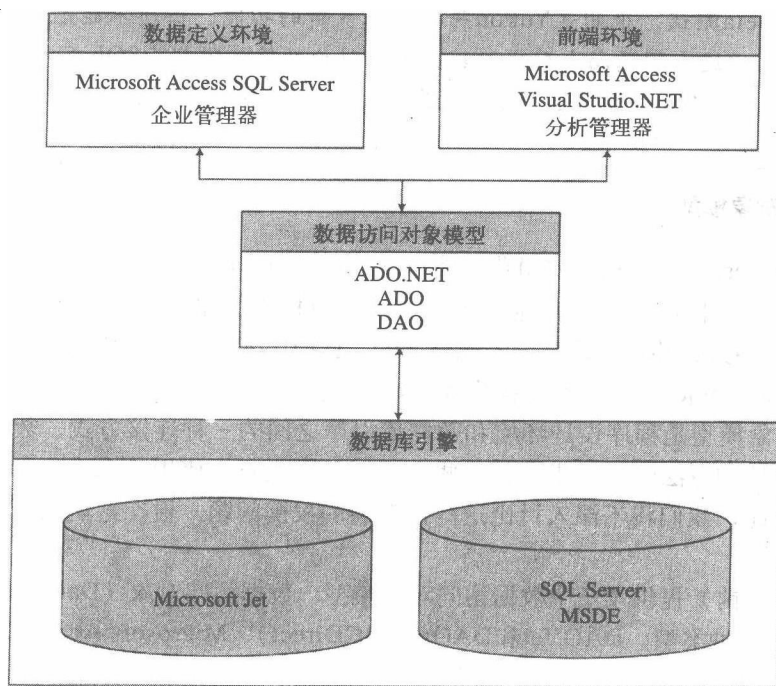


图1-2 本书所讨论的数据库工具

1.2.1 数据库引擎

在最底层的是**数据库引擎** (database engines), 有时也将数据库引擎称为“后端”, 但自从“后端”成为一个特定的物理架构 (我们将在第13章介绍) 的专有称呼之后, 这个词就显得不太合适了。这些工具将处理数据的物理操作——将数据存储到磁盘上, 并且在需要时将

其从磁盘中取出，我们将介绍两个引擎：Jet数据库引擎和SQL Server。你可能对Microsoft Access不是引擎感到很惊讶，从技术上来说，Access是一个前端开发环境，在这个环境中可以使用Jet引擎或SQL Server以及事实上任何ODBC兼容的数据库引擎作为它的数据存储。它使用Jet数据库引擎来操作存储在.mdb文件中的数据，使用SQL Server（或任何其他ODBC数据存储）来操作存储在.adp文件中的数据。Access总是使用Jet数据库引擎，直到Microsoft Visual Basic 3发布以前，Microsoft从没强调过它是一个独立的实体。

Jet数据库引擎和SQL Server尽管有很大差异，但都是存储和操作数据的非常好的工具。这两个引擎间的差别在于它们的架构以及它们面向的问题。Microsoft Jet是“桌面”型数据库引擎，用于小型到中型的系统（请注意这并不是说Jet数据库引擎只适合于价值不高的系统），而SQL Server使用客户/服务器架构，它面向中等规模到大规模的系统，它允许数千用户同时运行任务繁杂的应用程序。MSDE（Microsoft Desktop Engine的缩写）是SQL Server的简化版本，用于桌面型的使用。从设计者的角度看，在MSDE和SQL Server的完全版之间有一些小的差异，但我们这里并不考虑这些。在这本书中我们将介绍这两个数据库引擎之间的差异，并在第13章讨论在这两个架构间的权衡问题。

YUKON注意：我在2004年春写这本书的时候，SQL Server的一个新的版本——Yukon——正处在早期的Beta测试，很明显Yukon将包含很多新的功能，但还不能准确地说明这些改变是什么，由于这个问题尚未确定，我将主要讨论的范畴限制在SQL Server 2000的功能上，SQL Server 2000是目前已经发布的版本。如果某些功能可能有很大的变化时，我将在“YUKON注意”中说明。

1.2.2 数据访问对象模型

Microsoft Access，以及相对使用范围较小的Visual Studio.NET，提供了将窗体控件直接绑定到数据源的简单机制，使用户避免了直接处理数据库引擎。但是，在很多不同情况下（下面将讲到）这种方式并不总是可行的或合适的。此时，用户需要在代码中使用**数据访问对象模型**（data access object model）来操作数据。

数据访问对象模型是程序设计环境和数据库引擎之间的一种连接方式，数据访问控件模型提供了具有属性和方法的对象集合，这些方法可以在代码中使用。由于本书主要是讨论设计而非实现，因此，我们将不深入讨论这些模型间的权衡问题，但在这里回顾一下这些模型还是有好处的。

Microsoft（目前）提供了三个数据访问对象模型：数据访问对象（Data Access Objects，DAO，其中又有两种类型：DAO/Jet和DAO/ODBCDirect）、Microsoft ActiveX Data Objects（ADO）和ADO.NET。

DAO是三个当中历史最久的一个，它是Jet数据库引擎自身的接口。根据Microsoft 市场部的声明，DAO是在Microsoft Access环境中操作Jet数据库最有效的对象模型。ADO使用一个比DAO更简单的对象层次，它只由四个主要的对象组成，并且对模型提供了一些相当大的扩展——例如，它支持非连接的记录集和数据定制。它用于在Microsoft Access以及其他支持VBA的产品中操作ODBC兼容的数据库，包括Jet和SQL Server。ADO.NET是在.NET 框架中使用的ADO版本。

1.2.3 数据定义环境

Microsoft Jet和SQL Server是我们处理数据操作的物理层面，但我们需要一些方法来告诉它们如何结构化数据。Microsoft提供了很多的方法来实现这个功能，但我们只详细介绍其中的三个：对于关系模型介绍Access和SQL Server Enterprise Manager，对于维度模型介绍Analysis Manger。还有一些其他工具提供了几乎相同的功能，但这三个工具是我比较偏爱的。当然，一旦理解了这些原理，就可以使用对你的工作最有利的工具。

使用代码来定义你的数据库的结构也是可能的，并且我们将介绍如何实现它，尽管在通常的环境下并不推荐这种方式。除非由于某些原因你需要在运行时用程序来更改数据的结构（一种可能的例外情况是使用临时表，但我本人对这种做法持强烈的怀疑态度，如果数据库模式是不稳定的，那么你很可能没有理解问题域），否则，以上这些交互工具使用起来快速、简单并且也更加有趣。

1.2.4 前端开发

一旦制定好了数据库的物理定义，接下来，就需要用工具来创建用户将要交互的窗体和报表。我们将在Access和Visual Studio.NET（特别是Visual Basic.NET）两个环境中说明我们的例子，再强调一次，在我们的周围几乎有成百个前端工具可以使用，但设计原理是一样的，因此，可以将你在这里学到的知识应用到你所选的前端工具中。我们将在第10章简单地讨论一下Internet浏览器，但HTML本身超出了本书的范围。

1.3 关系模型

关系模型是基于数学理论，主要是集合论和谓词逻辑，这些原理首次被应用到数据模型的领域是在20世纪60年代末，是由IBM的研究者E.F.Codd博士于1970年提出的。关系模型的规则定义了数据可以被表达的方式（数据结构）、数据可以被保护的方式（数据完整性）以及在数据上可以实现的操作（数据操作）。

关系模型并不是能够存储和操作数据的唯一方法，还有其他模型：层次模型、网状模型和对象/数据模型。每一个这样的模型都有其拥护者，并且在某些方面都有其优势。但由于效率和灵活性的原因，关系模型是目前最广泛的数据库技术，并且也是本书讨论的内容。Microsoft Jet数据库引擎和Microsoft SQL Server 都是关系模型的。

我们也将讨论维度模型——一种特殊的关系模式，用于跟踪历史数据。我们将在第三部分介绍维度模型以及它与关系模型之间的关系。

一般来讲，关系数据库系统有如下特性：

- 所有的数据都被表达成行、列的有序集合，这个集合称为**关系**。
- 所有的值都是**确定的**，即对关系中任何给定的行、列位置，都有一个且仅有一个值。
- 所有的操作都是在关系中完成的，并且其结果也是一个关系，这个概念称为**闭包**。

如果你使用过Microsoft Access数据库，便会知道“关系”是一个“记录集 (recordset)”，在SQL Server术语中，关系是“结果集 (result set)”。Codd博士在阐明关系模型时，选择了术语“关系”而不使用“表”这个词，因为它相对来说内涵比较丰富。一个常见的误解认为之所以称之为是关系，是因为在表间建立了联系。事实上，这个名字是由其基于的关系所派

生而来的。

要注意的是，模型只要求数据在概念上表达为关系的，它并不指明数据应该如何被物理的存储。尽管现在看起来很明显，但在30年前当数据库程序设计通常意味着是编写机器代码来物理地操作数据存储设备时，这种概念上和物理上表达的区分是一个重大的革新。

事实上，关系并不需要有一个物理的表达，一个给定的关系可能映射到磁盘上的某个实际的物理表，但它也可能是从很多表中选出的列的综合，并且可以有计算列——计算列并没有物理地存储在任何地方，它只是由计算产生的。一个关系是以行和列的格式安排数据的表格，并且关系中的值都是确定的，它的存在完全独立于任何物理的表达。

要求关系中的每个值都是标量在某种程度上是不可靠的。“一个值”的概念是主观的，因为它是建立在数据模型中的语义基础上的。这里给出一个常见的例子，“名字”在一个数据模型中可能是一个单值，但在其他环境中可能需要将“名字”分解为“称呼”、“教名”和“姓氏”，而在另一个环境中可能又需要“中间名”和“尊称”。从严格意义上的术语来说，这些或多或少都有些不正确，因为它依赖于数据在什么地方使用。

基本表以及操作的结果都在概念上以关系的形式表达，这使得一个操作的结果可以作为另一个操作的输入。因此，不管是使用Jet数据库还是使用SQL Server，我们都可以将一个查询的结果作为另一个操作的基础。这为数据库设计者提供了在基于过程的开发中的功能上的相似性：封装复杂的或公共的操作并在任何需要的时候可以重用这些过程。

例如，你可能已经创建了一个叫做FullNameQuery的查询，这个查询将表达个人名字的不同属性组成一个叫做FullName的计算字段。你可以创建另一个查询，将FullNameQuery作为一个数据源，它对计算字段FullName的使用同基本表中的其他字段完全一样。因此，就没有必要再重新计算这个字段了。

当然，这是一个很简单的例子，并且很可能这个所谓的FullNameQuery并没有立即很明显地体现出比简单地重新计算姓名更有优势。但是，正如我们看到的，SQL语言允许生成相当复杂的查询，并且随着复杂程度的加深，使用已经存在的查询比输入新的语句要有利得多了。

1.4 关系术语

图1-3显示了一个标识了基本组件的正式名称的关系。了解一些关系设计的用户都会意识到这个关系不是规范化的。这没有关系，它仍然被限定为一个关系，因为它的数据是按行和列的形式排列的，并且它的值是确定的。

正如我们已经说过的，整个结构是一个关系，每个数据行是一个元组 (tuple)，从技术的角度来说，每一行是一个n-元组，但通常将“n-”省略。一个关系中元组的个数决定了它的基数 (cardinality)。在这个例子中，这个关系的基数是18。在元组中每个列称为是一个属性 (attribute)，在一个关系中属性的个数决定了它的度 (degree)。例子中的关系的度是3。

关系被划分为两个部分：标题 (heading) 和体 (body)。元组构成了关系的体，属性名构成了标题。注意，在这个例子的关系表达式中，每个属性的标签由两部分组成，中间用冒号分隔——例如，UnitPrice:Currency。标签的第一部分是属性的名字，第二部分是它的域。一个属性的域 (domain) 是数据被表达的“类型”，在这个例子中关系的域是：currency。

属性

SupplierName:CompanyName	ProductName:ProductName	UnitPrice:Currency
Pavlova, Ltd.	Alice Mutton	\$39.00
Plutzer Lebensmittelgroßmärkte AG	Thüringer Rostbratwurst	\$123.79
New Orleans Cajun Delights	Chef Anton's Gumbo Mix	\$21.35
G'day, Mate	Perth Pasties	\$32.80
Formaggi Fortini s.r.l.	Gorgonzola Telino	\$12.50
Specialty Biscuits, Ltd.	Sir Rodney's Scones	\$10.00
Tokyo Traders	Longlife Tofu	\$10.00
New Orleans Cajun Delights	Louisiana Hot Spiced Okra	\$17.00
Lyngbysild	Ragede sild	\$9.50
Grandma Kelly's Homestead	Northwoods Cranberry Sauce	\$40.00
Specialty Biscuits, Ltd.	Scottish Longbreads	\$12.50
Formaggi Fortini s.r.l.	Mascarpone Fabioli	\$32.00
Nord-Ost-Fisch Handelsgesellschaft mbH	Nord-Ost Matjeshering	\$25.89
Karkki Oy	Maxilaku	\$20.00
Svensk Sjöföda AB	Gravad lax	\$26.00
Exotic Liquids	Aniseed Syrup	\$10.00
Formaggi Fortini s.r.l.	Mozzarella di Giovanni	\$34.80
Heli Süßwaren GmbH & Co. KG	Gumbär Gummibärchen	\$31.23

← 标题

元组

体

图1-3 关系的组成元素

域同数据类型是不一样的，我们将在下一节详细讨论这个问题。域的说明一般不放在关系的标题中。

关系的体由0个或多个无序的元组组成，这里有一些重要的概念。首先，关系是无序的。想象一下一个关系就好比一个棕色的纸袋（或者你可以很诗意地将它想象为是一个明代瓷碗），其中包含了无特定次序的所有元组。记录号是在非关系型数据库中访问记录时使用的机制，但它并不应用在关系中。第二，一个没有元组的关系（空关系）依然等同于一个关系。第三，一个关系就是一个集合。根据定义，一个集合中的元素是唯一确定的。因此，对于符合某个关系的表，每一条记录必须是唯一确定的，即表中不包含重复的记录。

如果你阅读过Access或者SQL Server的文档，你可能会想为什么之前没有看过任何类似的文字。它们都是用于技术文献的正式术语，而不是只用于Microsoft的术语。我在这里介绍它们以便你不会在某个鸡尾酒会上感到尴尬（不管怎样，至少不会是因为有关3个度的n元组的话题）。

不幸的是，不仅Microsoft产品没有遵循正式的术语，而且它们本身还不一致。在表1-1中显示了分别用于两个产品中的术语。我们将在本书中使用正式术语。

表1-1 本书中用于产品检验的关系术语

正式术语		Microsoft Access	SQL Server
概念上的	物理上的		
关系	表	表或记录集	表或结果集
属性	字段	字段	列
元组	记录	记录	行

1.5 数据模型

数据库设计中最抽象的就是数据模型，即一个问题域的概念描述。数据模型是由实体、属性、域和联系来描述的。本章的剩余部分将依次分别讨论这些概念。

1.5.1 实体

很难给实体下一个精确的正式定义，但是这个概念已经是十分直白的：一个实体就是系统用来存储信息的事物。

当你开始设计数据模型时，确定一组最初的实体并不困难。当你（或者你的客户）谈论问题域时，大多数的名词和动词都将被用于候选实体。“Customers buy products. Employee sell products. Suppliers sell us products.”。名词“Customers”、“Products”、“Employees”以及“Suppliers”显然都是实体。

由动词“buy”和“sell”所描述的事件也都是实体，但是这里也存在一些问题。第一，动词“sell”被用来描述两个不同的事件：将产品卖给客户（Salesman → Customers）以及企业购买产品（Supplier → Company）。在这个例子中是很清楚，但这常常容易混淆，特别是当你问题域不熟悉的时候。

第二个问题与第一个问题刚好相反：两个不同的动词（第一个句子中的“buy”和第二个句子中的“sell”）被用来描述同一事件——客户购买产品。同样，其含义是不够清楚的，除非你对问题域十分熟悉。该问题比前一个问题更容易让人迷惑。如果一个客户在使用不同的动词来描述同一事件时，那么他们事实上可能是在描述不同类的事件。例如，如果该客户是一名裁缝，“customer buys suit”和“customer orders suit”可能都导致同样的结构——“sale of a suit”，但是在第一种情况下它是实时购买，而第二种它是预订购买。这显然是绝不相同的处理过程，当然也就需要在建模时区别开来。

除了在和客户面谈时创建一个实体列表之外，查看问题域中存在的任何文档也是十分有用的。输入表格、报表以及程序指南都是很好的候选实体的来源。但是，你必须对这些文档十分谨慎。打印的文档有很多限制——特别是输入表格，打印起来十分昂贵而且很少与规则和过程的变动保持一致。如果你困惑于某个实体从未在与客户的面谈中出现过，那么决不要猜想是该客户忘记提及。它很可能是一个已经过时的遗留元素，与公司并无任何关联。你需要去检验这一点。

大多数实体模型都在物理世界中有所映像：customers、products或者sales calls。这些都是**具体实体**。实体也可以对抽象概念建模。最常见的**抽象实体**的例子是在实体之间的联系上建立的建模——例如，某个销售代表负责某个客户，或者某个学生属于某个班级。

有时候，你需要建模的全部内容就是一个联系存在的事实。而有些情况下，你想要存储一些关于这个联系的额外信息，例如该联系建立的时间或者它的某些特征。美洲狮和丛林狼的关系就是竞争性的，而美洲狮和兔子之间则是捕食关系。如果你设计一个开放式的动物园的话，了解这些内容是非常有用的。

是否应当把那些没有属性的联系建模为单独的实体是需要进一步讨论的。我个人并不认为这么做可获得什么益处，并且它还使得从数据模型到数据库模式的转化过程变得更复杂了。但是，理解联系与实体同等重要是设计一个有效数据模型的准则。

1.5.2 属性

你的系统需要保存每个实体的一些事实。正如我们看到的，这些事实就是实体的属性。例如，如果你的系统包含一个Customers实体，那么你很可能想知道客户的名字和地址以及他

们所从事的职业。如果你正在给某个事件如ServicesCall建模，你很可能想知道客户是谁，哪些人打电话，什么时候打的，以及是否这些问题已被解决。所有这些都是属性。

决定是否将属性包含在你的模型中是一个语义过程。也就是说，必须将你的决定建立在这些数据的含义以及它们的使用方式的基础之上。我们来看一个普通的例子：一个地址。你会将地址建模为一个单独的实体（Address），还是作为一组实体（HouseNumber, Street, City, State, ZipCode）？大多数设计者（包括我自己）倾向于自动地将地址以一般原则分解为一组属性，这样结构化的数据会比较容易操作，但是这可能会不正确并且肯定不够直观。

让我们将一个本地业余音乐社团作为例子来说。希望存储它的会员的地址以便打印邮寄标签。这是该地址被输入的唯一目的，所以这里就不必将一个地址看作是单个多行的文本，这是完全在需求范围之外的。

但是，如果是一个所有业务都在互联网上进行的邮寄订购公司，那又该如何呢？为了销售税金的目的，该公司需要知道他的客户所居住的州。它可能从该音乐社团使用的单个文本字段中抽取州，这不太容易；所以在这种情况下单独将州作为一个属性就十分有意义了。那么剩余的地址部分如何处理呢？你可能会想用一组属性（HouseNumber, Street, City, State, ZipCode）比较合适。但是那样的话你需要处理单元住宅号、邮局信箱号以及陆军邮局（APO）地址。又怎样处理转交给其他人的地址？当然世界变得越来越小，但其复杂性并没有减少，所以当你得到在美国之外的第一个客户的时候又会发生什么呢？美国的地址遵循一套相当标准的模式，但是当你开始处理国际订单的时候情况就不是这样的了。

你不仅需要知道国家并且调整邮政编码的格式，而且属性的排列可能也需要改变。例如，在欧洲的大部分地区，房屋的号码跟在街道名称后面。这不是很糟糕，当你输入的时候将其映射过来就可以了，但是有多少美国数据录入人员能够清楚在地址4/32 Griffen Avenue, Bondi Beach, Australia中，4/32的含义是Apartment 4, Number 32。

这里的关键不是那些地址有多难建模，尽管它们是比较难，真正困难的是，你不能对应怎样对任何特定类型的数据建模作任何假设。你为处理国际邮件订购开发的复杂模式肯定不能适用于本地音乐社团的地址。

著名的艺术家马蒂斯说过，一幅绘画作品，当它不应当有任何添加和删减时才意味着它的完成。实体设计有一点与此类似。如何得知你达到这一点了呢？不幸的回答是你可能永远不知道（并且，即使你认为做到了，它也很可能随着时间改变）。在当今的技术条件下，没有任何途径可以开发一个完全正确的数据库设计。你可以证明一些设计是有瑕疵的，但是你不能证明任何给定的设计没有缺陷。如果你愿意，你是无法证明自己的无辜的。你怎样应付这样的问题呢？这里没有规则，但是有一些策略。

第一个策略是：**从结果出发，不要将设计做得比它实际需要的复杂。**你的数据库必须回答什么问题？在我们的第一个例子中，对于那个音乐社团，唯一的问题是：“我写信给这个人的时候应该寄到哪里？”所以一个单一属性模型就足够了。第二个例子中，对于邮寄订购公司来说，需要回答的问题只是“这个人生活在哪个州？”，因此我们需要一个不同的结构来提供这个结果。

当然，当你试图提供灵活的方式，它不仅可以处理你的用户正在询问的问题，还能回答你预见他们将来可能询问的问题时，就需要十分仔细了。例如，我敢打赌那个音乐社团系统实现一年后，社团就会回来要求你可以按邮政编码来对地址排序，这样他们就能方便处理批

量的邮购折扣。

你也应当小心用户问这样的问题，如果他们仅仅知道他们可以问问题的话，特别是当你使一个手工系统自动化的时候。想象一下，询问一个图书管理员在这四百万册书籍中有多少是1900年前在芝加哥出版的，她或他会将你指向卡片文件并告诉你在哪找答案。然而对一个设计优良的数据库系统来说这却是一个微不足道的问题。

好的设计者的特征之一就是周到和创意，凭借这些他们能发现潜在的问题。常常听没有经验的分析者埋怨用户不知道他们想要的。他们当然不知道，帮助他们发现他们想要的是分析者的工作。

这导致了第二个策略：**找寻例外情况**。这个策略有两重含义。第一，你必须确定所有的例外情况；第二，你必须设计系统尽可能多地处理例外情况，以便不使用户迷惑。为了说明这个含义，让我们来看另外一个例子：个人名字。

如果你的系统将被用来产生相应的通信，那么很关键的一点是使名字正确无误。（例如，那些主动提供的寄给Mr.R.M.Riordan的邮件甚至都没有被打开过。）大多数名字都是很清楚。Ms.Jane Q.Public是由Title、FirstName、MiddleInitial以及LastName组成的，对吗？错。（你是这么看的，对吗？）首先，FirstName和LastName是文化的特定效果。更保险的（并且从官方角度看更正确的）是使用GivenName和Surname。其次，Sir James Peddington Smythe和Lord Dunstable会是怎样的情况呢？Peddington Smythe是他的Surname吗？或者Peddington是他的MiddleName吗？另外你如何分解“Lord Dunstable”？歌手Sting又该如何？那是一个GivenName还是一个Surname？曾经作为王子的艺术家该如何处理呢？你确实关心这个吗？

这最后一个问题并不像它看起来的那样轻率。一封寄给Sir James Peddington Smythe的信可能不会冒犯任何人。但是，存在问题的不是Sir Smythe；可能招来问题的是Sir James，或者可能是Lord Dunstable。虽然，从现实意义上讲，你的客户中有多少是领主呢？本地音乐社团不会感激你给他们一个界面如图1-4所示的会员系统。

请记住，灵活性和复杂性之间存在一个平衡。虽然尽可能找到更多的例外情况是很重要的，但是也有足够的理由排除一些例外，因为它们不太值得花太大成本去处理。

The image shows a screenshot of a web form for an address. It is highly complex with many input fields and labels. The labels on the left side include: Courtesy Title, Given Name, Middle Name(s), Surname, Degrees & Titles, Company Name, Care of, Salutation, Apartment/Unit, House Number, Building Number, Street 1, Street 2, City, State or Province, ZIP, Postal Office, Floor, Suburb, and Country. There are also checkboxes for 'Address as "Title"' and 'Salute'.

图1-4 一个过于复杂的地址界面

区分实体和属性有时候是比较困难的。地址就是一个很好的例子，同样，你的决定必须建立在问题域的基础之上。一些设计者提倡在系统建模时创建一个单一的地址实体来存储所有的地址。从实现的角度来看，这个方法在封装性和代码重用方面有一定的优点。但从设计者的角度来看，我对这种设计有所保留。

比如，雇员的地址和客户的地址不太可能使用同样的方式。例如，大量雇员的邮件基本上都是通过内部邮件而不是邮政服务来处理的。在这种情况下，规则和要求都是不同的。图1-4中显示的数据输入界面或者一些类似的界面可能比较适合客户地址的记录，但是使用一个单一地址实体，会被强制在管理雇员地址时也使用它，而这可能是没有必要的，也是不合适的。

1.5.3 域

你可能回想起本章开头部分，每个关系标题都包含一个属性名:域名对 (AttributeName: DomainName)。我们那时说过一个域定义指定了属性所描述的数据类别。更确切地说，一个域是一个属性可能包含的所有有效值的集合。

域常常与数据类型相混淆；它们是不同的。数据类型是一个物理概念，而域是一个逻辑概念。“Number”是一个数据类型；“Age”是一个域。再看另一个例子，“StreetName”和“Surname”可能都描述为文本字段，但是它们显然是不同类别的文本字段；它们属于不同的域。

相对数据类型来说，域是一个较为狭窄的概念，域定义包含了关于有效数据的更多的特定描述。以DegreeAwarded为例，它描述了一所大学所授予的学位。在数据库模式中，该属性可能被定义为Text[3]，但它不是任意一个3个字符长的字符串。它是集合{BA, BS, MA, MS, PHD, LLD, MD}中的一个元素。

当然，并不是所有的域都能由简单地列出它们的值来定义。例如，当我们谈论人的时候，Age包含了大约100个值，但是如果我们谈论博物馆的展出品，它则有成千上万个取值。在每个例子中都可以使用规则来约束有效值，而不需要罗列所有的值简单的定义域。例如，PersonAge可以定义为“一个范围在0到120之间的整型数”，而ExhibitAge可以简单的定义为“一个大于等于0的整型数”。

在这一点上，你可能考虑域是数据类型和有效规则的组合。当然，如果以这种方式考虑，你可能不会错得太远。但是严格来讲，有效性规则是数据完整性的部分，而不是数据描述的部分。例如，一个邮政编码的有效规则可能是参考State属性，而邮政编码的域是“一个有5位数字的字符串”（或者如果你使用邮政分区+4的格式，它可能是一个11位的字符串）。

注意，每一个这样的定义都与存储的数据类别（number或者string）相关联。这很像一种数据类型，但实际上不是。正如我们说过的，数据类型是物理的；他们通过数据库引擎来定义和实现。将一个域定义为varchar (30) 或者Long Integer是错误的，因为这是具体引擎所描述的。

对于任意两个域来说，如果将它们所定义的属性进行比较是有意义的话（并且进一步，在它们之间作关系操作，例如连接操作，我们将在第5章讨论这部分内容），那么这两个域被认为是“类型兼容”的。例如，在图1-5给出的两个关系中，将它们通过EmployeeID = SalespersonID连接起来是十分有意义的。比如，你可以通过这样的操作来获取给定雇员的发货单。那么EmployeeID和SalespersonID两个域之间就是类型兼容的。但是，试图通过EmployeeID = OrderDate将两个关系连接起来则很可能得到毫无意义的结果，即使这两个域被定义为同样的数据类型。

Order ID	Customer	Salesperson ID	Order Date
10643	Alfreds Futterkiste	6	25-Aug-1997
10692	Alfreds Futterkiste	4	03-Oct-1997
10702	Alfreds Futterkiste	4	13-Oct-1997
10835	Alfreds Futterkiste	1	15-Jan-1998
10952	Alfreds Futterkiste	1	16-Mar-1998

Employee ID	Title Of Courtesy	Given Name	Surname	Title
1	Ms.	Nancy	Davolio	Sales Representative
2	Dr.	Andrew	Fuller	Vice President, Sales
3	Ms.	Janet	Leverling	Sales Representative
4	Mrs.	Margaret	Peacock	Sales Representative
5	Mr.	Steven	Buchanan	Sales Manager
6	Mr.	Michael	Suyama	Sales Representative
7	Mr.	Robert	King	Sales Representative
8	Ms.	Laura	Callahan	Inside Sales Coordinator
9	Ms.	Anne	Dodsworth	Sales Representative

图1-5 雇员和订购关系

不幸的是，Jet 数据库引擎和SQL Server都没有给域提供超过数据类型的强大的内部支持。并且即使是在数据类型里；这两个引擎也没有执行足够的检查；二者都是默默地在幕后转换数据的。例如，如果你使用Microsoft Access并且已经在Employee表中定义EmployeeID为长整型，在Invoices表中定义InvoiceTotal为货币型的值，那么，你可以创建一个通过EmployeeID = InvoiceTotal将两表连接起来的查询，并且Microsoft Jet将十分高兴地给你一个所有EmployeeID与发货单总值相匹配的雇员名单。这两个属性不是类型兼容的，但是Jet数据库引擎并不知道也不关心这一点。

那么为什么要如此重视域呢？我们将会在本书的第三部分看到，因为它们是十分有用的设计工具。“这两个属性可以互换吗？”，“有没有规则只能应用到其中一个而不能应用到其他的？”。在你设计一个数据模型的时候，这些都是十分重要的问题，并且域分析可以帮助你考虑这些问题。

1.5.4 联系

除了每个实体的属性之外，一个数据模型还必须指定实体之间的联系。从概念层面上来讲，联系就是实体之间简单的关联。语句“Customers buy products”表明实体Customers和Products之间存在的联系。包含在联系中的实体被称为“参与者”。参与者的数目就是该联系的“度”。（联系的度和关系的度类似，但不完全相同，后者是指属性的个数。）

大部分的联系都是二元的，例如“Customers buy products”，但这并不是一种硬性要求。三元联系有三个参与者，也是较为常见的。在给定的二元联系“Employees sell products”和“Customers buy products”中存在一个隐含的三元联系“Employees sell products to customers”。但是，指定两个二元联系并不能使我们确定哪位雇员将哪些产品销售给哪些客户；只有三元联系能做到这一点。

二元联系中的一个特别的例子是某个实体与其自身参与了一个联系。这常常被称作原料清单关系，并且常用来描述级别结构。这种联系的一个常见例子是雇员和经理：任何给定的雇员可能本身就是一位经理，而同时又隶属于某位经理。

两个实体之间的联系可能是一对一、一对多或者是多对多的。一对一的联系很少见，但是在某些环境下是十分有用的。

一对多联系可能是最常见的类型。一张发货单包含多种产品。一位销售人员可以创建多张发货单。这些都是一对多联系例子。

虽然不如一对多联系那样常见，但是多对多联系也并不罕见并且也存在很多实例。客户购买很多产品，而产品也可以由多个客户购买。老师教多个学生，而学生也是由多个老师来教的。在关系模型中不能直接实现多对多联系，但是可以很方便的间接实现它们。这些我们将在第3章中看到。

一个联系中任何给定的实体参与者可以是局部的也可以是全局的。如果某个实体不参与任何联系就不可能存在的话，那么这个实体者就是全局的；否则，它就是局部的。例如，销售员在信息逻辑上是不存在的，除非有与之对应的雇员。反之则不成立。一个雇员可能是其他角色而并不一定是销售员，所以一个雇员可以存在而不需要对应的销售员记录。因此，该联系中的雇员参与者是局部的，而销售员参与者是全局的。

这里的难题是如何确保指定的局部或全局参与者对实体的所有实例总是正确的。例如，对于公司来说，如果要改变某种产品的供应商，但在“Suppliers provide products”关系中的Products参与者被定义为全局的，那么在不删除产品信息的情况下是无法删除当前供应商的。

1.5.5 实体联系图

实体联系模型是以实体、属性和联系来描述数据的，它是由Peter Pin Shan Chen 在1976年提出的。同时，他提出了一种图形方法称为实体联系（E/R）图，该方法被广泛接受。E/R图使用矩形来描述实体，用椭圆描述属性，用菱形描述联系，如图1-6所示。

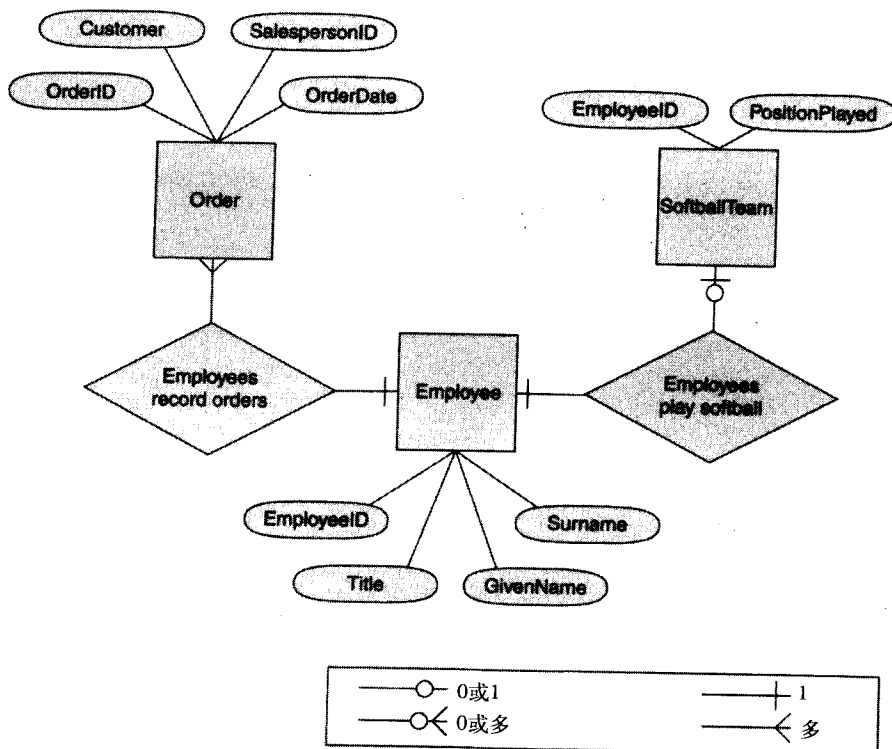


图1-6 一个E/R图

实体间联系的特征（一对一、一对多或者多对多）用不同的方式来表示。一些人使用1和M或者1和 ∞ （代表无穷）来表示一和多。我们使用“鸟爪”状符号技术，如图1-6所示。这种方法更具表达力。

E/R图的最大优点是它们容易画也容易理解。在实际应用中，经常将属性分开来画，因为它们存在的细节程度不同。一般来说，人们或者考虑模型中的实体以及它们之间的联系，或者考虑一个给定实体的属性，但是很少同时考虑二者。

1.6 小结

在这一章中，我们探讨了数据库系统的组件并为本书的剩余部分打下了基础。我们从说明问题域是作为现实世界中已定义的某个部分开始本章，然后介绍概念数据模型是用实体、属性和它们所定义的域来描述问题域。数据模型的物理层是数据库模式，数据库模式是以数据库的形式展示出来的。最后，数据库引擎实现应用程序对数据库的物理操作，而应用程序是由窗体和报表组成，并与用户交互。

在下一章中，我们将学习数据库结构的细节，并探讨规范化的原则。

第2章 数据库结构

这一章将讨论设计关系型数据库模型的第一个方面：关系自身的结构。该设计阶段的主要目标很简单：确保模型能够回答任何合理的问题。第二个目标是最小化数据冗余以及与之相关的问题。我们将先讨论后者，然后再讨论第一个问题。

2.1 消除冗余

数据冗余浪费资源，并且更重要的是，它让操作变得非常的困难。以图2-1所显示的记录集为例，它描述的是一个公司的出货单。（假设此时，该记录集就是明确存储在数据库中的一张基本表，而不是一个查询的结果。）

Order ID	SalesPerson	Hire Date	Phone	Company Name	Product Name	Quantity
10871	Dodsworth, Anne	15-Nov-1994	452	Bon app'	Alice Mutton	16
10747	Suyama, Michael	17-Oct-1993	428	Piccolo und mehr	Gorgonzola Telino	8
10258	Davolio, Nancy	01-May-1992	5467	Ernst Handel	Chef Anton's Gumbo Mix	65
11007	Callahan, Laura	05-Mar-1994	2344	Princesa Isabel Vinhos	Thüringer Rostbratwurst	10
10421	Callahan, Laura	05-Mar-1994	2344	Que Delícia	Perth Pasties	15
10558	Davolio, Nancy	01-May-1992	5467	Around the Horn	Perth Pasties	18
10431	Peacock, Margaret	03-May-1993	5176	Bottom-Dollar Markets	Alice Mutton	50
10659	King, Robert	02-Jan-1994	465	Queen Cozinha	Gorgonzola Telino	20
10273	Leverling, Janet	01-Apr-1992	3355	QUICK-Stop	Gorgonzola Telino	15
10362	Peacock, Margaret	03-May-1993	5176	Ernst Handel	Chef Anton's Gumbo Mix	32
10949	Fuller, Andrew	14-Aug-1992	3457	Bottom-Dollar Markets	Alice Mutton	6
10286	Davolio, Nancy	01-May-1992	5467	QUICK-Stop	Perth Pasties	36
10867	Suyama, Michael	17-Oct-1993	428	Lonesome Pine Restaur	Perth Pasties	3
10691	Fuller, Andrew	14-Aug-1992	3457	QUICK-Stop	Thüringer Rostbratwurst	40
10354	Callahan, Laura	05-Mar-1994	2344	Pericles Comidas clásic	Thüringer Rostbratwurst	4
10698	Peacock, Margaret	03-May-1993	5176	Ernst Handel	Thüringer Rostbratwurst	12
10962	Callahan, Laura	05-Mar-1994	2344	QUICK-Stop	Perth Pasties	20
10465	Davolio, Nancy	01-May-1992	5467	Vaffeljernet	Thüringer Rostbratwurst	18
10549	Buchanan, Steven	17-Oct-1993	3453	QUICK-Stop	Gorgonzola Telino	55

图2-1 存在冗余数据的记录集

你会发现每位雇员的HireDate和Phone两列值都被罗列了数次。这将会产生一系列的后果。首先，这意味着每当输入一条新出货信息时，都不得不再次输入这两个字段的值。并且，每当你输入信息时就多了一次出现错误的可能。例如，如图2-2所示的记录集，你如何得知Nancy Davolio是在1992年还是在1999年被聘用的？

她什么时候被雇用的？

Order ID	SalesPerson	Hire Date	Phone	Company Name	Product Name	Quantity
10871	Dodsworth, Anne	15-Nov-1994	452	Bon app'	Alice Mutton	16
10747	Suyama, Michael	17-Oct-1993	428	Piccolo und mehr	Gorgonzola Telino	8
10258	Davolio, Nancy	01-May-1992	5467	Ernst Handel	Chef Anton's Gumbo Mix	65
11007	Callahan, Laura	05-Mar-1994	2344	Princesa Isabel Vinhos	Thüringer Rostbratwurst	10
10421	Callahan, Laura	05-Mar-1994	2344	Que Delícia	Perth Pasties	15
10558	Davolio, Nancy	01-May-1999	5467	Around the Horn	Perth Pasties	18
10431	Peacock, Margaret	03-May-1993	5176	Bottom-Dollar Markets	Alice Mutton	50

图2-2 重复数据会导致不一致性

第二，在一名新雇员有销售记录之前，这样的结构会阻止你存储新雇员的聘用日期和电话。第三，如果一条出货信息已经建档并从数据库中删除，那么相关的聘用日期和电话号码信息就会丢失。

这样的问题通常被称作更新异常，如果冗余数据被存储在多个关系中，则问题将会更加严重。考虑图2-3中显示的记录集。（同样，假设这是基础表而不是查询结果。）如果Alfreds Futterkiste的电话号码改变了，那么需要在7个不同的地方进行修改——1个在Customer记录集中，6个在出货记录集中。

Customers关系

Customer ID	Company Name	Phone
ALFKI	Alfreds Futterkiste	030-0074321
ANATR	Ana Trujillo Emparedados y helados	(5) 555-4729
ANTON	Antonio Moreno Taquería	(5) 555-3932
AROUT	Around the Horn	(171) 555-7788
BERGS	Berglunds snabbköp	0921-12 34 65
BLAUS	Blauer See Delikatessen	0621-08460
BLONP	Blondel père et fils	88.60.15.31
BOLID	Bólido Comidas preparadas	(91) 555 22 82
BONAP	Bon app'	91.24.45.40
BOTTM	Bottom-Dollar Markets	(604) 555-4729
BSBEV	B's Beverages	(171) 555-1212
CACTU	Cactus Comidas para llevar	(1) 135-5555
CENTC	Centro comercial Moctezuma	(5) 555-3392

Invoices关系

Order ID	Company Name	Phone
10643	Alfreds Futterkiste	030-0074321
10692	Alfreds Futterkiste	030-0074321
10702	Alfreds Futterkiste	030-0074321
10835	Alfreds Futterkiste	030-0074321
10952	Alfreds Futterkiste	030-0074321
11011	Alfreds Futterkiste	030-0074321
10308	Ana Trujillo Emparedados y helados	(5) 555-4729
10625	Ana Trujillo Emparedados y helados	(5) 555-4729
10759	Ana Trujillo Emparedados y helados	(5) 555-4729
10926	Ana Trujillo Emparedados y helados	(5) 555-4729
10365	Antonio Moreno Taquería	(5) 555-3932
10507	Antonio Moreno Taquería	(5) 555-3932
10535	Antonio Moreno Taquería	(5) 555-3932
10573	Antonio Moreno Taquería	(5) 555-3932
10677	Antonio Moreno Taquería	(5) 555-3932

图2-3 同样的重复数据可能存在于多个记录集中

这并不是不可能或者难于做到的事情，关键的问题是很难记住这么多要修改的地方。并且，即使不会忘记任何事情，又如何能确保维护人员在六个月后更新系统时知道存在这种类型的冗余，更不可能让他们记得（或知道）怎样恰当地处理它们。较好的方式是回避这些冗余并将产生的问题都归结到一起。

但是，你需要确定这些你认为的冗余属性确实是冗余的。看看图2-4所示的例子。初看之下，可能会认为在这两个关系中的UnitPrice属性是冗余的。但是它们实际上是代表不同的两个值。在Products关系中的UnitPrice属性代表是当前销售价，在Order关系中的UnitPrice属性表示的是该产品被销售时的价格。例如，Tofu在Orders关系中的UnitPrice被标为\$18.60，而在Products关系中被标为\$23.25。Tofu现在卖\$23.25的事实不能改变它在过去某个时刻被售卖为\$18.60的事实。这两个属性被定义为相同的域，但是它们逻辑上是不同的。

Product ID	Product Name	Unit Price
1	Chai	\$18.00
2	Chang	\$19.00
3	Aniseed Syrup	\$10.00
4	Chef Anton's Cajun Seasoning	\$22.00
5	Chef Anton's Gumbo Mix	\$21.35
6	Grandma's Boysenberry Spread	\$25.00
7	Uncle Bob's Organic Dried Pears	\$30.00
8	Northwoods Cranberry Sauce	\$40.00
9	Mishi Kobe Niku	\$97.00
10	Ikura	\$31.00
11	Queso Cabrales	\$21.00
12	Queso Manchego La Pastora	\$38.00
13	Konbu	\$6.00
14	Tofu	\$23.25

这些值并不相同

Order ID	Product Name	Unit Price	Quantity	Unit Price
10248	Mozzarella di Giovanni	\$34.80	5	\$174.00
10248	Queso Cabrales	\$21.00	12	\$168.00
10248	Singaporean Hokkien Fried Mee	\$14.00	10	\$98.00
10249	Manjimup Dried Apples	\$53.00	40	\$1,696.00
10249	Tofu	\$18.60	9	\$167.40

图2-4 同样标识的数据可能并不是冗余的

2.2 保证灵活性

这一设计阶段的另一目标是保证数据库的灵活性——它能够回答所有合理的问题。灵活性主要是由它的完备性（显然没有一个数据库系统能够提供它没有包含的数据）以及它的结构决定的。但是回答该问题的简易方式几乎毫无疑问就是结构的结果。这里的原则是易于将属性和关系组合起来，而拆开它们则是十分困难的。

Employee ID	Title	Given Name	Surname	Title
1	Ms.	Nancy	Davolio	Sales Representative
2	Dr.	Andrew	Fuller	Vice President, Sales
3	Ms.	Janet	Leverling	Sales Representative
4	Mrs.	Margaret	Peacock	Sales Representative
5	Mr.	Steven	Buchanan	Sales Manager
6	Mr.	Michael	Suyama	Sales Representative
7	Mr.	Robert	King	Sales Representative
8	Ms.	Laura	Callahan	Inside Sales Coordinator
9	Ms.	Anne	Dodsworth	Sales Representative

Employee ID	FullName
1	Ms. Nancy Davolio, Sales Representative
2	Dr. Andrew Fuller, Vice President, Sales
3	Ms. Janet Leverling, Sales Representative
4	Mrs. Margaret Peacock, Sales Representative
5	Mr. Steven Buchanan, Sales Manager
6	Mr. Michael Suyama, Sales Representative
7	Mr. Robert King, Sales Representative
8	Ms. Laura Callahan, Inside Sales Coordinator
9	Ms. Anne Dodsworth, Sales Representative

图2-5 连接信息容易，但从复合字段中抽取信息就比较困难

例如，给定如图2-5所示的两个关系，FullName可以很容易的用以下语句从第一个关系中得到：

```
TitleOfCourtesy & " " & GivenName & " " & Surname & _
    ", " & Title
```

但是从第二个关系中的FullName字段中只检索LastName将需要对该字符串自身进行操作:

```
Function GetLastname(FullName) As String
    Dim lastname As String

    'strip off the Title
    lastname = Left(FullName, InStr(FullName, ",") - 1)
    'strip off the TitleOfCourtesy
    lastname = Right(lastname, Len(lastname) - _
        InStr(lastname, " "))
    'strip off FirstName
    lastname = Right(lastname, Len(lastname) - _
        InStr(lastname, " "))

    GetLastname = lastname

End Function
```

这样的技术很容易被FullName字段内容的多变性破坏。例如,名字“Bill Rae Jones”将会返回“Rae Jones”,而你可能只想得到“Jones”。产生一个LastName格式的列表,FirstName可能会变得十分难看。

在创建数据模型时涉及的第二个原则可以有效地回答被查询的问题,以避免在回答查询时要求评价来自多个字段的相同信息的情况。比如,以图2-6中所显示的关系为例,两者都是为学生注册建模。

StudentID	GivenName	Surname	Period1	Period2	Period3
1	Nancy	Davolio	Biology	History	English
2	Andrew	Fuller	Physical Education	Biology	French
3	Janet	Levering	Physical Education	Biology	French
4	Margaret	Peacock	French	Physical Education	History
5	Steven	Buchanan	Biology	French	Physical Education
6	Michael	Suyama	French	Physical Education	Biology
7	Robert	King	History	French	English

StudentClassID	GivenName	Surname	Period	Class
10	Margaret	Peacock	1	French
19	Robert	King	1	History
4	Andrew	Fuller	1	Physical Education
16	Michael	Suyama	1	French
7	Janet	Levering	1	Physical Education
13	Steven	Buchanan	1	Biology
1	Nancy	Davolio	1	Biology
2	Nancy	Davolio	2	French
5	Andrew	Fuller	2	Biology
8	Janet	Levering	2	Biology
11	Margaret	Peacock	2	Physical Education
14	Steven	Buchanan	2	French
17	Michael	Suyama	2	Physical Education
20	Robert	King	2	French
21	Robert	King	3	Biology
9	Janet	Levering	3	French
15	Steven	Buchanan	3	Physical Education
6	Andrew	Fuller	3	French
18	Michael	Suyama	3	Biology
3	Nancy	Davolio	3	English
12	Margaret	Peacock	3	History

图2-6 这两个记录集包含相同的数据

为了回答问题“哪些学生今年修Biology?”，使用第一个关系，你可能不得不在三个字段中查询值“Biology”。对应的SQL SELECT语句如下：

```
SELECT StudentID FROM Enrollments WHERE Period1 = "Biology"
OR Period2 = "Biology" OR Period3 = "Biology"
```

如果使用第二个结构，则只需要查询一个字段：Class：

```
SELECT StudentID FROM Enrollments WHERE Class = "Biology";
```

这两种方式都能工作，但显然第二个方法要更简单更不容易出现编码错误，自然也更容易想到。

避免冗余并且使检索数据更容易是数据建模中需要了解的全部内容，剩下的只是尝试规范化这两个基本原则。但是如果你曾经作了很多（或任何）数据建模工作，那么你会清楚它们是很简单的，这些原则可以很自然的应用起来。它们好比一个纸张夹：一旦你看到它，答案就很明显了，但是在你第一次面对一大堆散落的纸盒和一小段金属线时，提出这样的解决方法确实有点困难。

2.3 基本原则

在问题域中结构化数据的过程要达到两个目的：消除冗余和保证灵活性，该过程被称作规范化。这一章的剩余部分将讨论规范化的原则，它是用来控制数据的结构的工具，就好比纸张夹控制一张张纸一样。范式（我们将讨论六种）为关系的结构逐一深入的规定了要求。每一种范式都是前一种的扩展，用这种方式来防止某种形式的更新异常。

需要切记的是范式并不是创建“正确的”数据模型的万能良药。一个数据模型可能被彻底的规范化了，但是仍旧不能回答某些问题；或者，它可能提供答案，但是速度太慢并且非常的笨拙，使得数据库系统是不可用的。但是如果你的数据模型是规范化的——也就是说，如果它遵循关系结构的规则——那么其成为一个有效的数据模型的机会就会高很多。

不过，在我们关注规范化之前，应该熟悉下述原则。

2.3.1 无损分解

关系模型允许关系以不同的形式通过属性连接起来。获得一个完全规范化的数据模型的过程就是通过以某种方式分解关系来消除冗余，而分解之后的关系又能重性组合起来而不丢失任何信息。这就是无损分解原则。例如，在图2-7显示的关系中，你可以将其分解为如图2-8所示的两个关系。

Order ID	Order Date	Required Date	Customer ID	Company Name	Address	City
10853	27-Jan-1998	24-Feb-1998	BLAUS	Blauer See Delikatessen	Forsterstr. 57	Mannheim
10905	24-Feb-1998	24-Mar-1998	WELLI	Wellington Importadora	Rua do Mercado, 12	Resende
10953	16-Mar-1998	30-Mar-1998	AROUT	Around the Horn	120 Hanover Sq.	London
11016	10-Apr-1998	08-May-1998	AROUT	Around the Horn	120 Hanover Sq.	London
10736	11-Nov-1997	09-Dec-1997	HUNGO	Hungry Owl All-Night Grocers	8 Johnstown Road	Cork
11022	14-Apr-1998	12-May-1998	HANAR	Hanari Carnes	Rua do Paço, 67	Rio de Janeiro
10577	23-Jun-1997	04-Aug-1997	TRAJH	Trail's Head Gourmet Provisioners	722 DaVinci Blvd.	Kirkland
10750	21-Nov-1997	19-Dec-1997	WARTH	Wartian Herkku	Torikatu 38	Oulu
10324	08-Oct-1996	05-Nov-1996	SAVEA	Save-a-lot Markets	187 Suffolk Ln.	Boise

图2-7 一个非规范化关系

Customer ID	Company Name	Address	City
ALFKI	Alfreds Futterkiste	Obere Str. 57	Berlin
ANATR	Ana Trujillo Emparedados y helados	Avda. de la Constitución 2222	México D.F.
ANTON	Antonio Moreno Taquería	Mataderos 2312	México D.F.
AROUT	Around the Horn	120 Hanover Sq.	London
BERGS	Berglunds snabbköp	Berguvsvägen 8	Luleå
BLAUS	Blauer See Delikatessen	Forsterstr. 57	Mannheim
BLONP	Blondel père et fils	24, place Kléber	Strasbourg
BOLID	Bólido Comidas preparadas	C/ Araquil, 67	Madrid

Order ID	Order Date	Required Date	Customer
11077	36-May-1998	03-Jun-1998	RATTC
11076	36-May-1998	03-Jun-1998	BONAP
11075	36-May-1998	03-Jun-1998	RICSU
11074	36-May-1998	03-Jun-1998	SIMOB
11073	35-May-1998	02-Jun-1998	PERIC
11072	35-May-1998	02-Jun-1998	ERNSH
11071	35-May-1998	02-Jun-1998	LILAS
11070	35-May-1998	02-Jun-1998	LEHMS
11069	34-May-1998	01-Jun-1998	TORTU

图2-8 图2-7中的关系能被分解成这两个关系并且不丢失任何信息

2.3.2 候选码和主码

在第1章中，我已将一个关系的主体定义为一个包含0个或多个元组的无序集合，并且指出根据定义，集合中的每个元素都是唯一的。在这种情况下，对于任何关系都必然存在某些属性的组合，它们唯一确定每一个元组。如果数据行不能被唯一确定，它们组合元组的方式就不符合关系理论。这里一个或多个属性的结合被称作**候选码**。

对任意给定的关系，可能存在多个候选码，但是每个候选码都必须能够唯一标识每个元组，而不仅仅是对某些特定的元组集合，它应该总能适用于所有可能的元组。并且，该原则的逆命题也必须能够成立。给定任意两个有相同候选码的元组，它们必须表达的是相同的实体。

与关系设计的很多事宜一样，一个候选码的确定是语义上的。你不能通过观察来确定。如果一些字段或者一些属性的组合只对某个给定的元组集是唯一的，那么你不能保证它对所有的元组都是唯一的，但这是作为一个候选码的必要条件。同样，你必须明白数据模型的语义——数据的含义是什么，而不是它表面的意思。

考虑图2-8中下面的Orders关系。在这个例子中CustomerID是唯一的；但是它不可能一直保持如此。毕竟，大多数公司依靠的是重复的业务。尽管表面上是唯一的，但模型的语义告诉我们该字段不能成为一个候选码。

根据定义，所有关系必须至少有一个候选码：所有属性的集合组成了元组。候选码可以由单个属性组成（称为**简单码**），或者由多个属性组成（称为**复合码**）。

一些人认为复合码在某种程度上是不正确的，必须通过在表中添加一个人工的标识符——一个确定的或者随机数字段——来避免这种复合码的情况。没有事情是绝对错误的。正如我们看到的，人为标识符有时确实很方便，但是复合码也完全可以接受。

然而，候选码的另一个要求是它是不能再化简的，因此所有属性的集合没有必要成为一个候选码。在图2-9所示的关系中，属性{CategoryName}是一个候选码，{Description}也是，但是集合{CategoryName, Description}，虽然也是唯一的，但它并不是一个候选码，因为Description属性是没有必要的。

Category Name	Description
Beverages	Soft drinks, coffees, teas, beers, and ales
Condiments	Sweet and savory sauces, relishes, spreads, and seasonings
Confections	Desserts, candies, and sweet breads
Dairy Products	Cheeses
Grains/Cereals	Breads, crackers, pasta, and cereal
Meat/Poultry	Prepared meats
Produce	Dried fruit and bean curd
Seafood	Seaweed and fish

图2-9 候选码必须是不可化简的，因此{ CategoryName }是候选码，
而{ CategoryName, Description }则不是

有时候会出现这样的情况——虽然不是经常发生——一个关系可能存在多个可能的候选码。在这种情况下，需要指定其中的一个候选码作为主码，而其他的候选码作为备用码。这是一个很武断的决定，而且在逻辑层面上没什么用处。（记住数据模型是纯粹抽象的。）为了帮助维护模型和它物理实现之间的差别，我倾向于在数据模型级别使用术语“候选码”，而在实现时使用“主码”。

当唯一可能的候选码不实用时——比如，它要求太多的属性或者太大——则可以使用一个人工的唯一字段数据类型来创建人造码，其值可以由系统产生。

只要你不需要人造码有任何含义，那么Microsoft Jet中的随机数（AutoNumber）字段以及SQL Server中的标识（Identity）字段，都是实现人造码的十分有用的工具。这些字段就是一个标志而已。它们不保证是顺序的，你也基本不能控制它们是如何生成的，并且如果你试图使用它们来表达某些含义的话，其所带来的麻烦可能比你已解决的问题还要多。

正如我们看到的，虽然选择候选码是一个语义过程，但不要假设你用来确定现实世界实体的那些属性可以成为合适的候选码。例如，人们通常是凭借他们的名字来引用的，但是浏览任何电话簿，你会发现名字是很难唯一的。

当然，在与其他属性组合时，名字可以作为候选码，但是这将会很难决定。我曾经和20位同事在一个办公室工作过，其中有两位叫Larry Simon，有一位叫Lary Simon。这三位都是副主管。在我们之间，他们被称作“Short Lary”，“German Larry”和“Blond Larry”；这是身高、民族和头发颜色与名字的组合，但很难作为一个可行的候选码。

在这些情形下，最好是使用系统生成的ID数，比如随机数或者标识符字段，但是要记住，不要试图让它们有任何含义。当然，你需要十分小心你的用户会故意添加一些明显的重复项。但正如我们将在第20章中看到的，最好是将其作为用户界面的一部分来处理而不是给系统维护的数据强加一个人为（也完全没有必要）的约束。

2.3.3 函数依赖

在考虑数据结构时，函数依赖的概念确实是十分有用的工具。给定一个元组T，有两个属性集合 $\{X_1, \dots, X_n\}$ 和 $\{Y_1, \dots, Y_m\}$ （这两个集合不需要是相互排斥的），如果对于任意合法的X值，仅有唯一合法的Y值与之对应，那么就称集合Y函数依赖于集合X。

例如，在图2-9所示的关系中，每一个{ CategoryName }值相同的元组都有相同的{ Description }值。因此我们可以说属性CategoryName函数决定属性Description。注意函数依赖不能以其他方式工作：知道一个Description的值不会允许我们确定相应的CategoryID值。

你可以通过图2-10所示的形式来表明属性集合之间的函数依赖。在本书中，可以将函数

依赖表示为 $X \rightarrow Y$ ，读作“X函数决定Y”。

学术界对函数依赖是十分感兴趣的，因为它提供了一种使数据建模类似于数学的机制。例如，只要你愿意，就可以讨论关于一个函数依赖的自反性和传递性。

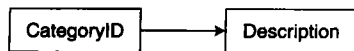


图2-10 函数依赖图都是不言自明的

在实际应用中，函数依赖是表述一个非常清晰的概念的一种方便的方法：给定任意关系，总会存在一些属性对每个元组都是唯一的，并且知道这些属性就能决定那些非唯一的属性。

因此，给定元组 $\{X, Y\}$ ，如果 $\{X\}$ 是一个候选码，那么所有的属性 $\{Y\}$ 必须函数依赖于 $\{X\}$ ；这是由候选码的定义决定的。如果 $\{X\}$ 不是一个候选码，并且函数依赖很重要（也就是说， $\{Y\}$ 不是 $\{X\}$ 的子集），那么该关系很可能涉及一些冗余，并且需要进一步规范化。再看图2-9所示的例子，知道了CategoryName的值，我们就能很容易的确定CategoryDescription。

在某种意义上，数据规范化是一个确保所有箭头都是由候选码发出的过程，如图2-10所示。

2.4 第一范式

如果某个关系的属性所定义的域均是标量，那么称该关系是第一范式的。这个概念曾经是数据建模中最简单也是最难的概念之一。该原则可以直白地表述为：一个元组的每个属性都仅包含一个单一值。但是一个单一值是由什么组成的呢？在图2-11所示的关系中，Items属性显然包含多个值，因此它不是第一范式的。但是问题并不是总是如此清楚的。

OrderID	CustomerID	OrderDate	Items	OrderTotal
1	CACTU	1/1/1999	3 Zaanse koeken, 1 Tarte au sucre	\$89.70
2	BSBEV	1/5/1999	4 Mozzarella di Giovanni	\$139.20
3	SUPRD	5/2/1999	3 Ravioli Angelo, 6 Tofu	\$198.06

图2-11 该关系中的Items属性不是标量

我们看一下第1章中对名字和地址的建模过程，就会发现在决定一个属性是否是标量的问题上，我们常常会遇到一些麻烦。日期是另一个麻烦的域。它们有三个不同的组件：日、月和年。应当将其存储为三个属性还是一个组合体呢？通常，答案是依据你建模的问题域的语义来决定。

如果你的系统大部分时候会将这三个组件作为一个整体数据来使用，那么就将其作为标量。但是如果你的系统经常操作日期的单个组件，那么你最好是将它们存储为单个的属性。例如，你可能不关心日，而仅关心月和年。或者你可能只关心月和日，而不在意年。这种情况很少，但它确实存在。

在具体的日期实例中，由于日期的计算实现起来比较复杂，如果你使用某个属性将其定义为日期时间类型，那么会使操作较容易一些。日期时间型的数据是将日期的三个组件以及时间组合在一起了。日期时间数据类型在开发环境中可以减轻用户的不少负担，比如计算从今天起之后第37天的日期。

然而，如果你在比较其中单个组件时，日期时间属性可能会给你带来麻烦，特别是当你忽略了该字段的时间组件的时候。例如，如果你是使用VBA的函数Now来设置DateCreated字段的值，该函数将返回一个当前的日期以及时间。而之后你却试图将它与一个由Date()产生的返回值进行比较，这个函数仅返回日期，那么你可能会得到无法预料的结果。即使你从不再

时间显示给用户，但它们仍然存储在了数据中，“1/1/1999 12:30:19 AM”与“1/1/1999”显然是不同的。

除此之外，人们经常会在非标量值产生问题的地方是编码和标志位。很多公司都将事件编码或参照编码设置为一个计算值，时常会看到类似REF0010398的数据行，它可能标明第一个事例是在1998年三月开始的。当然你不可能改变公司的规定，但是这样在你的数据模型中如果要操作参照编码的每个部分则会很不方便。

将该值分开来存储会变得容易很多：{Reference#, Case#, Month, Year}。这样，决定下一个事件编码或者事件在某个给定年的开始编码将会变成一个针对某个属性的简单查询，而不需要额外的操作。这有重要的性能寓意，特别是在客户/服务器环境下，从一个属性的中间抽取一个值可能要在客户端检查每一条记录（并且通过网络来传送），而不是在数据库服务器端进行。

另一种给人们造成问题的非标量属性类型就是位标志。在传统的编程环境下，将布尔值作为一个字中的一个位来存储是十分普遍的做法，然后就可以使用位逻辑运算来检查和测试它们。例如，Windows API编程就十分依赖这种技术。在传统编程环境中，这么做是一件十分有意义的事情。但在关系数据模型中，则不是了。这种操作不仅违反了第一范式的要求，而且作为一个通用规则，它变得十分繁重和没有效率。

不幸的是，由于历史的某些原因，这成为一种强加给你的约束，但是如果在这样的环境下你有任何其他的选择，请不要在单个属性中将其编码为超过一个的信息数。如果你在使用遗留的信息，可以在记录集中将数据分解并存储两个版本。

当检查第一范式的关系时，需要警惕另一种非标量值：重复组。图2-12显示了一个发货关系。在某种程度上，一些人决定客户不可以购买三件以上的产品。我怀疑他们是否首先与销售经理检查过该决定。显然，这不是业务要求，而仅仅是由系统强加的一个人为约束。人为的系统约束是很糟糕的，在上述情况中，它显然也是错误的。

另一个重复组的例子如图2-13所示。这不是一个明显的错误，并且使用类似的模型已经实现了很多成功的系统。但是这确实是图2-12结构的另一种形式，存在相同的问题。想象一下这样的查询：查询在所有年份的第一季度的销售额超过目标的10%的产品。

OrderID	CustomerID	Item1	Qty1	Item2	Qty2	Item3	Qty3
1	ANTON	Queso Cabrales	4	Tofu	3	Ravioli Angelo	1
2	BLAUS	Chai	2		0		

图2-12 该数据模型限制客户购买产品的数量

Product	Year	TargetJan	ActualJan	TargetFeb	ActualFeb
Aniseed Syrup	2004	\$1,000.00	\$1,300.00	\$0.00	\$0.00
Chai	2004	\$4,000.00	\$2,000.00	\$0.00	\$0.00
Chang	2004	\$3,000.00	\$8,022.00	\$0.00	\$0.00

图2-13 这是一个重复组

2.5 第二范式

如果一个关系是第一范式的，并且它的所有属性都完全依赖于候选码，那么称该关系是第二范式的。例如，图2-14所示的候选码是{ProductName, SupplierName}，但是SupplierPhoneNumber字段只依赖SupplierNumber，而不是全部的复合码。

Product Name	SupplierName	Category Name	SupplierPhoneNumber
Chai	Exotic Liquids	Beverages	(171) 555-2222
Chang	Exotic Liquids	Beverages	(171) 555-2222
Guaraná Fantástica	Refrescos Americanas LTDA	Beverages	(11) 555 4640
Sasquatch Ale	Bigfoot Breweries	Beverages	(503) 555-9931
Steeleye Stout	Bigfoot Breweries	Beverages	(503) 555-9931
Côte de Blaye	Aux joyeux ecclésiastiques	Beverages	(1) 03.83.00.68
Chartreuse verte	Aux joyeux ecclésiastiques	Beverages	(1) 03.83.00.68
Ipoh Coffee	Leka Trading	Beverages	555-8787
Laughing Lumberjack Lager	Bigfoot Breweries	Beverages	(503) 555-9931
Outback Lager	Pavlova, Ltd.	Beverages	(03) 444-2343

图2-14 关系中所有的属性都应当依赖于整个候选码

我们已经看到这种情况会造成冗余现象，并且这种冗余会转变成不愉快的维护问题。一种更好的模型如图2-15所示。

Product ID	Product Name	Category
1	Chai	Beverages
2	Chang	Beverages
3	Aniseed Syrup	Condiments
4	Chef Anton's Cajun Seasoning	Condiments
5	Chef Anton's Gumbo Mix	Condiments
6	Grandma's Boysenberry Spread	Condiments
7	Uncle Bob's Organic Dried Pears	Produce

Supplier ID	SupplierName	SupplierPhoneNumber
1	Exotic Liquids	(171) 555-2222
2	New Orleans Cajun Delights	(100) 555-4822
3	Grandma Kelly's Homestead	(313) 555-5735
4	Tokyo Traders	(03) 3555-5011
5	Cooperativa de Quesos 'Las Cabras'	(98) 598 76 54
6	Mayumi's	(06) 431-7877
7	Pavlova, Ltd.	(03) 444-2343
8	Specialty Biscuits, Ltd.	(161) 555-4448
9	PB Knäckebröd AB	031-987 65 43

图2-15 这两个关系是第二范式的

从逻辑上来讲，这是一个是否将两个不同的实体：Products和Suppliers，描述为同一个关系的问题。如果分别描述，不仅消除了冗余，并且也提供了一种存储信息的机制，即不可能通过其他途径获得这些信息。在图2-15所示的例子中，你可以获取有关Suppliers的信息而不需要考虑与他们相关的产品的任何信息。这在第一种关系中是无法做到的，因为主码的任意一个成分都不可以为空。

另一个有关第二范式的麻烦是人们会对下面这种约束感到迷惑：该约束在给定的某个时刻是正确的，但它并不是一直都是正确的。例如，在图2-16所示的关系中，假定一个供应商只有一个地址，这在某个时刻可能是正确的，但是将来无法保证它一直是正确的。

Supplier ID	Address	City	Region	Postal Code
1	49 Gilbert St.	London		EC1 4SD
2	P.O. Box 78934	New Orleans	LA	70117
3	707 Oxford Rd.	Ann Arbor	MI	48104
4	9-8 Sekimai	Tokyo		100
5	Calle del Rosal 4	Oviedo	Asturias	33007
6	92 Setsuko	Osaka		545
7	74 Rose St.	Melbourne	Victoria	3058
8	29 King's Way	Manchester		M14 6SD
9	Kaloadagatan 13	Göteborg		S-345 67
10	Av. das Americanas 12.890	São Paulo		5442

图2-16 一个供应商可能有多个地址

2.6 第三范式

如果一个关系是第二范式的，并且所有非主码属性都是相互独立的，那么就称该关系是第三范式的。让我们来看一个公司的例子，每个公司在各个州都有唯一的一位销售员。给定如图2-17所示的关系，在Region和Salesperson之间存在一种依赖关系，但是这两个属性都不合适作为该关系的候选码。

Order ID	Company Name	Region	Salesperson
11076	Bon app'	WA	Margaret Peacock
11077	Rattlesnake Canyon Grocery	WA	Nancy Davolio
11075	Richter Supermarkt	WA	Laura Callahan
11074	Simons bistro		Robert King
11071	LILA-Supermercado	WA	Nancy Davolio
11072	Ernst Handel	WA	Margaret Peacock

图2-17 尽管Region和Salesperson相互独立，但它们都不适合做候选码

很可能第三范式过于学术了。例如，多数情况下，你会依据City和Region来决定PostalCode的值，因此图2-18所示的关系并不是严格的第三范式。

图2-19所示的两个关系在技术上更正确一些，但是在现实中，你可以从中获得的唯一好处是当你输入新记录时能自动查询PostalCode值，从而让用户少敲几下键盘。这虽然不是一个微不足道的好处，但是完全可以通过其他的方式来实现该功能，而不需要在每次引用地址时都做关系之间的连接。

Company Name	Address	City	Region	Postal Code
Exotic Liquids	49 Gilbert St.	London		EC1 4SD
New Orleans Cajun Delights	P.O. Box 78934	New Orleans	LA	70117
Grandma Kelly's Homestead	707 Oxford Rd.	Ann Arbor	MI	48104
Tokyo Traders	9-8 Sekimai	Tokyo		100
Cooperativa de Quesos 'Las Cabras'	Calle del Rosal 4	Oviedo	Asturias	33007
Mayumi's	92 Setsuko	Osaka		545
Pavlova, Ltd.	74 Rose St.	Melbourne	Victoria	3058

图2-18 该关系并不是严格意义上的第三范式

Company Name	Address	City	Region
Exotic Liquids	49 Gilbert St.	London	
New Orleans Cajun Delights	P.O. Box 78934	New Orleans	LA
Grandma Kelly's Homestead	707 Oxford Rd.	Ann Arbor	MI
Tokyo Traders	9-8 Sekimai	Tokyo	
Cooperativa de Quesos 'Las Cabras'	Calle del Rosal 4	Oviedo	Asturias
Mayumi's	92 Setsuko	Osaka	
Pavlova, Ltd.	74 Rose St.	Melbourne	Victoria

City	Region	Postal Code
Melbourne	Victoria	3058
Ste-Hyacinthe	Québec	J2S 7S8
Montréal	Québec	H1J 1C3
Bend	OR	97101
Sydney	NSW	2042
Ann Arbor	MI	48104
Boston	MA	02134
New Orleans	LA	70117
Oviedo	Asturias	33007

图2-19 这两个关系是第三范式

如同数据建模过程中的其他任何决定一样，何时以及怎样实现第三范式仅能通过考虑模型的语义来决定，不可能给出固定的规则，但是有一些指南：你应该创建一个独立的关系，当且仅当

- 该实体对于模型来说十分重要，或者
- 数据变更很频繁，或者
- 你确定存在技术实现优势

邮政编码会变更，但并不频繁；并且他们在大多数系统中并不绝对重要。另外，在大多数现实世界的应用中，定义一个单独的邮政编码表是不实际的，因为对于如何定义邮编存在不同的规则。

2.7 进一步的规范化

前三种范式都已包含在Codd关于关系理论的原文档中，并且在绝大多数情况下，它们是你所有需要关心的问题。只要记住我在高校里学的一句俗话：“主码，所有的码，除了码外别无其他，Codd，请帮我一把！”。

进一步的范式——Boyce/Codd范式、第四范式和第五范式——都是为了处理特殊情况而开发出来的，它们大多很少见。

2.7.1 Boyce/Codd范式

Boyce/Codd范式是与第三范式不同类型的，用来处理有多个候选码的关系的情形。事实上，对于Boyce/Codd范式的应用，必须满足以下条件：

- 该关系必须拥有两个或多个候选码。
- 至少两个候选码必须是复合的。
- 候选码必须有重叠的属性。

理解Boyce/Codd范式最简单的方式是使用函数依赖。Boyce/Codd范式主要说明了在候选码之间不能存在函数依赖。以图2-20所示的关系为例来说，该关系是第三范式的（假定供应商的名称是唯一的），但是它始终存在严重的冗余。

Supplier ID	SupplierName	Product	Quantity	Unit Price
5	Cooperativa de Quesos 'Las Cabras'	Queso Cabrales	12	\$14.00
20	Leka Trading	Singaporean Hokkien Fried Mee	10	\$9.80
14	Formaggi Fortini s.r.l.	Mozzarella di Giovanni	5	\$34.80
24	G'day, Mate	Manjimup Dried Apples	40	\$42.40
6	Mayumi's	Tofu	9	\$18.60
24	G'day, Mate	Manjimup Dried Apples	35	\$42.40
19	New England Seafood Cannery	Jack's New England Clam Chowder	10	\$7.70
2	New Orleans Cajun Delights	Louisiana Fiery Hot Pepper Sauce	15	\$16.80

图2-20 该关系是第三范式但不是Boyce/Codd范式的

在上述情况中，候选码是{SupplierID, ProductID}和{SupplierName, ProductID}，它们之间的函数依赖图如2-21所示。

正如你看到的，存在这样一个函数依赖{SupplierID} → {SupplierName}，这与Boyce/Codd范式是相违背的。正确的模型应该如图2-22所示。

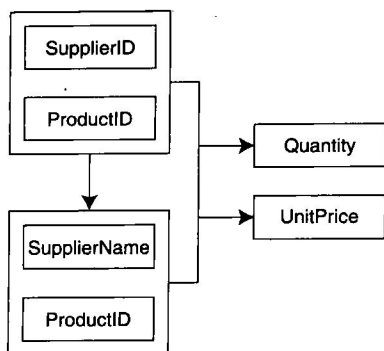


图2-21 图2-20所示关系的函数依赖示意图

SupplierID	SupplierName
1	Exotic Liquids
2	New Orleans Cajun Delights
3	Grandma Kelly's Homestead
4	Tokyo Traders
5	Cooperativa de Quesos 'Las Cabras'
6	Mayumi's

SupplierID	ProductID	Quantity	UnitPrice
2	65	15	\$21.05
24	53	15	\$32.80
8	20	40	\$81.00
22	47	16	\$9.50
6	14	9	\$23.25
28	59	30	\$55.00
28	60	40	\$34.00
21	46	15	\$12.00

图2-22 这是图2-21所示关系完全规范化之后的模型

2.7.2 第四范式

第四范式为一个直观的原则提供了理论基础：独立的重复组不应当被组合在一个关系中。在例子中，我们假定由Northwind Traders销售的自身品牌的产品使用不同规格的包装，而它们又是来自不同的供应商，并且所有的供应商提供所有包装规格。一个完全非规范化的Products关系版本可能就同图2-23所示的一样。

ProductName	SupplierName	PackSize
Chai	Exotic Liquids	8 oz., 16 oz., 32 oz.
Chef Anton's Cajun Seasoning	New Orleans Cajun Delights	8 oz., 16 oz., 32 oz.
Pavlova	Pavlova, Ltd.	8", 10"

图2-23 这是一个非规范化的关系

现在，规范化该关系的第一步是消除非标量的PackSize属性，结果如图2-24所示。

意外的是，图2-24所示的关系是符合Boyce/Codd范式的，因为它是“全主码”的。但是该关系明显存在冗余问题，并且维护数据的完整性也十分困难。这些问题的解决方法就依赖“多值依赖对”的概念和第四范式。

ProductName	SupplierName	PackSize
Chai	Exotic Liquids	8 oz.
Chai	Exotic Liquids	16 oz.
Chai	Exotic Liquids	32 oz.
Chef Anton's Cajun Seasoning	New Orleans Cajun Delights	8 oz.
Chef Anton's Cajun Seasoning	New Orleans Cajun Delights	16 oz.
Chef Anton's Cajun Seasoning	New Orleans Cajun Delights	32 oz.
Pavlova	Pavlova, Ltd.	8"
Pavlova	Pavlova, Ltd.	10"

图2-24 这是图2-23所示关系的Boyce/Codd范式版本

一个多值依赖对是两个相互独立的属性集合。在图2-23中，多值的依赖是{ProductName} → {PackSize} || {SupplierName}，该依赖可以读作“Product 多值决定 PackSize和Supplier”。从非正式的角度来说，第四范式表明多值依赖必须分解为不同的关系，如图2-25所示。正式来说，如果一个关系是Boyce/Codd范式，并且所有的多值依赖同时也是函数依赖于候选码，则其就是第四范式的。

ProductName	PackSize
Chai	32 oz
Chef Anton's Cajun Seasoning	32 oz.
Chai	16 oz.
Chef Anton's Cajun Seasoning	16 oz.
Pavlova	10"
Chai	8 oz.
Chef Anton's Cajun Seasoning	8 oz.
Pavlova	8"

ProductName	SupplierName
Chai	Exotic Liquids
Chai	Exotic Liquids
Chai	Exotic Liquids
Chef Anton's Cajun Seasoning	New Orleans Cajun Delights
Chef Anton's Cajun Seasoning	New Orleans Cajun Delights
Chef Anton's Cajun Seasoning	New Orleans Cajun Delights
Pavlova	Pavlova, Ltd.
Pavlova	Pavlova, Ltd.

图2-25 包含多值依赖的关系必须被分解

2.7.3 第五范式

第五范式用来处理极少有的“连接依赖”情况。一个连接依赖描述的是一种循环约束“如果实体1与实体2连接，实体2与实体3连接，而实体3反过来又与实体1连接，并且这三个实体都必须存在于相同的元组中”。

为了将其说得更通俗易懂一些，我们看一下这个示例，如果{Supplier}供应{Product}，{Customer}订购{Product}，并且{Supplier}提供某种东西给{Customer}，那么就说{Supplier}提供{Product}给{Customer}。然而，在现实世界中这并不是一个有效的推断。{Supplier}可以提供任何东西给{Customer}，不一定必须是{Product}。当且仅当一个额外的约束表明该推断是有效的时，才说明存在一个连接依赖。

在该示例中，使用一个具有属性{Supplier, Product, Customer}的关系是不够的，因为这样会产生更新问题。例如，给定如图2-26所示的关系，插入一个元组{“Ma Maison”, “Aniseed Syrup”, “Berglunds snabbkop”}，则需要插入第二个元组，{“Exotic Liquids”, “Aniseed Syrup”, “Berglunds snabbkop”}，因为一个新的由“Aniseed Syrup”到“Berglunds snabbkop”的连接已经被添加到模型中了。

Supplier	Product	Customer
Exotic Liquids	Aniseed Syrup	Alfreds Futterkiste
Exotic Liquids	Chef Anton's Cajun Seasoning	Berglunds snabbkop

图2-26 这个关系不采用第五范式

将关系分解为三个不同的关系（SupplierProduct、ProductCustomer以及SupplierCustomer）可以消除该问题，但也会引起其自身的问题；如果重建原关系，则所有这三个关系都必须连接起来。暂时仅连接其中的两个关系将会导致无效的信息。

从一个系统设计人员的角度来说，这是一个十分可怕的情形，因为除了通过安全限制之外，没有内在的机制可以使得三张表连接起来。此外，如果一个用户创建一个临时的结果集，该结果看上去十分合理，那么用户就不太可能通过观察发现错误。

幸运的是，这种循环的连接依赖十分少见，大多数情况下都可以安全地将其忽略不计。当它们不能忽略时，唯一的方法就是构建一个数据库系统，在其中提供确保多关系连接的完整性的措施。

2.8 小结

在本章中，我们通过规范化的过程，学习了数据库的结构。规范化中最基本的原则是通过无损分解来消除冗余——即在不丢失信息的前提下将关系进行分解。该原则用正式的术语来说就是范式。前边的三个范式是经常应用的，它们包含在俗语“主码，所有的码，除了码外别无其他，Codd，请帮我一把！”中。剩余的三种范式只在特定的意外情形下使用。

下一章，我们将探讨当我们探索实体间关系的建模时，如何将关系逻辑地连接起来。

第3章 联 系

第2章我们探讨了数据库结构的设计，这是一个对问题域中的实体进行分析，并在此基础上开发一套高效且有力地获取所有相关数据关系的过程。但是关系仅仅是数据模型的一部分。这些关系之间的关联以及这些关联上的约束也是同样重要的。在这一章，我们将关注**联系**的建模，即关系模型中关联的表达。正如定义关系一样，一旦你理解了数据模型的语义，那么，其基本的原则就十分简单了。当然会有某些特殊的情况不完全适合联系的模型，我们会在后边讨论这些情况。

3.1 术语

这里有一些应用于联系的基本术语。那些相关联的实体称为**参与者**，在一个联系中参与者的数目称为联系的**度**。绝大多数的联系是二元的，它有两个参与者，但是一元联系（一个与其自身相关联的关系）也很普遍，而三元联系（有三个参与者）也不是不存在。本章所涉及的大多数联系都是二元的。在本章的后面我们将把一元和三元联系作为特殊情况来学习。

在一个联系中的参与者可以被分为**全局的**或**局部的**，划分的依据是该实体是否独立于联系而存在。例如，给定两个实体Customer和Order，此联系中Customer参与者是局部的，因为Customer的细节信息可以在客户做任何订购前被单独输入到系统中。而另一方面，Order则是全局参与者，因为如果没有Customer是不可能做任何订购的。

有时候可以使用同样的原则将实体本身划分为**弱实体**（是全局参与者）和**常规实体**（是局部参与者）。弱实体仅当与其他实体有联系时才能存在，而常规实体可以独立存在。正如其开创者Peter Pin Shan Chen所描述的那样，这种划分方式是实体联系图方法的一部分。你可以使用图3-1所示的符号来表明一个实体是弱实体还是常规实体。

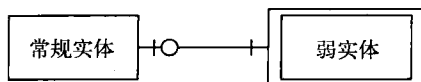


图3-1 该符号能被用来区分弱实体和常规实体

在一个联系中参与者的这种划分也是联系**可选性**的一种表现：一个实体是否被要求参与某个给定的联系。这是一个难办的问题，因为数据库引擎实现的并不与该问题域相匹配，当我们在第4章讨论数据完整性的实现时会看到这一点。

有时候将联系划分为“IsA”和“HasA”两种类型。这个概念是很清晰的：实体X要么IsA实体Y，要么HasA实体Y。举个例子，一名雇员“是一个”（IsA）垒球队的成员；同样的雇员可以“有一个”（HasA）地址。当然，“是”和“有”在描述一个联系时并不总是很好的自然语言术语。一名雇员没“拿到”一个销售订单，他“创建”了一个销售订购，但是这显然不能说雇员“是”一个销售订单，这种智能的延伸并不太理想。

一个实体的实例能够被另一个实体的某个实例所引用的最大数目称为一个联系的**基数**。（注意，度和基数在应用到关系和联系中都有不同的含义。）联系的基数有三种类型：一对一，一对多以及多对多。

我们使用图3-2所示的符号来表明联系的基数和可选性。在第一章中引入的鸟爪状的符号

是最简单也是最具描述力的给客户的解释方式。当然，也存在其他的技术，可以选用最适合你的工作的方法。

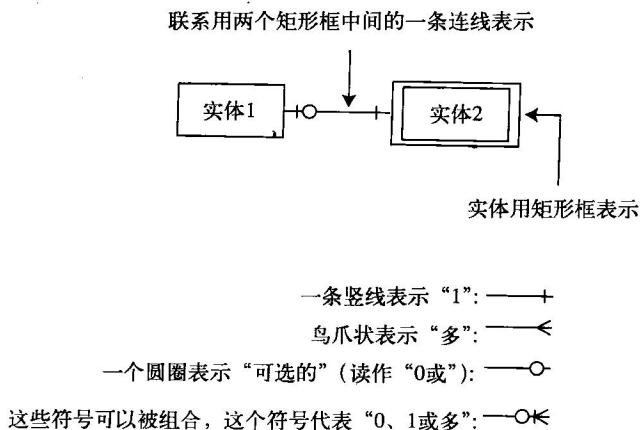


图3-2 该符号用来表明可选性和基数

3.2 联系建模

一旦确定了某个联系的存在，那你必须通过在其中一个关系（外部关系）中包含另一个关系（主关系）中的属性的方法来构建该联系，如图3-3所示。

主关系和外部关系的选择不是随意的。它首先是由联系的基数来决定的，其次是依据数据模型的语义决定的（这种情形很少，但无疑它是存在的）。例如，给定两个一对多联系的关系，处在“一”端的关系总是主关系，而在“多”端的关系总是外部关系。也就是说，处在“一”端关系中的候选码被添加（作为外码）到“多”端的关系中。当我们在本章的剩余部分探讨各种类型的联系时会说明这个问题。

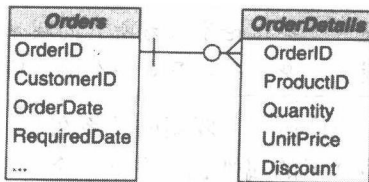


图3-3 通过在外部关系（OrderDetails）中包含主关系（Orders）属性来对联系进行建模

绘制联系

你会注意到图3-3和图1-6所示的正式的E/R图有所区别。首先，属性没有作为独立的对象显示。在这种设计层面上，你主要关心的是实体间的联系，而不是它们的组成。单独显示属性会让关注点分散，并且使得图形较为散乱。

第二，联系没有被标记。一些分析者（以及一些绘图工具）在每个联系中都添加一个描述标签。在我们的例子中，该标签可能是类似“Orders consist of OrderDetails”的一句话。但是标签是没有必要的，因为联系的描述会根据你阅读它的方向而改变（老师教学生，而学生向老师学习），标签时常会混淆这些。

正如我所说的，这种方式的图形在与客户交流时会十分有用，并且通过手绘或者使用绘图工具如Microsoft Visio都能十分容易地画出这些图形。但是Microsoft Access、Microsoft SQL Server以及Microsoft Visual Studio都提供了绘图工具，并各有特点，可以根

据我所讲述的技术来选择使用你喜欢的工具。

使用Access关系窗口（对于Jet数据库引擎“.mdb”文件）或者使用数据库图形（对于SQL Server实现的数据库）的优点在于图形成了数据库的一部分，并且能自动的反映数据库的变化。不幸的是，这也同时是这些工具的缺陷。你不能创建抽象图形，而必须创建物理表格。在实现模型最终定案之前，在设计阶段过早的进入到实现时的任务，都是很危险的。

在自身的工作中，我经常同时使用抽象图形和嵌入在数据库中的图形。我在设计过程的早期阶段创建抽象图形，并且在概念设计最终完成后使用微软的某一工具来策划物理数据库模式。

当然，并不能随意的将主关系中的属性拷贝到外部关系中，而是必须选择那些唯一标识主实体的属性。换句话说，可以将那些主关系中组成候选码的属性添加到外部关系中。毫无疑问，这些复制的属性将成为外部关系中的外码。在图3-3所示的例子中，OrderID——Orders关系的候选码——已经被添加到OrderDetails关系中了。Orders是主关系，OrderDetails是外部关系。

注意，构建联系的一对候选码/外码不一定是主表中的主码；任何候选码都可以使用。应当选用最能表达语义含义的候选码。

有时候，不仅要某个联系存在的事实建模，而且要对联系的某些属性建模——比如联系的持续周期或者它的开始时间。在这种情况下，创建一个用来表达联系的抽象关系是十分有用的。在图3-4中所显示的Positions关系就是一个关于“联系的关系”的例子。

注意：一些理论家认为所有的联系都必须单独建模，但是依我看来他们的讨论总是围绕联系模型本身，而不是对一个问题域建模的常规方法。

分开描述表达联系的关系在某种程度上使得数据模型更加复杂，而用户可能只是简单地试图将联系的属性包含在某一个参与者关系中。这种方法其实并没有如此糟糕，但是如果存在很多属性，或者属性间有很多关联，它将变得十分笨拙。更重要的是，一个不同的联系实体允许你跟踪一个联系的历史。例如，图3-4中所示的模型允许你确定每个的雇员的历史，但如果Position是Employees关系的一个属性，则这就不太可能了。

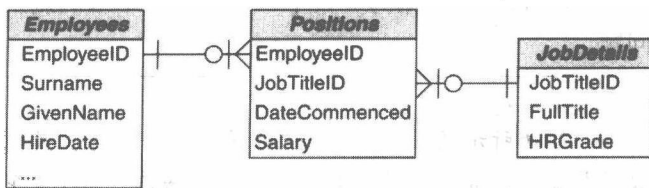


图3-4 抽象关系可以对联系的属性建模

当你需要跟踪一个联系随时间变化的方式时，抽象联系实体也十分有用。例如，图3-5是一个状态转变图，它描述的是个人婚姻状态的所有可能的合法变化。

状态转变图理解起来并不困难。每个垂直线代表了一个合法的状态，而水平线则描述了一个状态到另一个状态的变化。例如，某个人可以从“结婚（married）”到“离婚（divorced）”，但是不能从“离婚（divorced）”到“未婚（never married）”。

现在，如果所有需要你做的就是对个人的婚姻状态进行建模，那么就不需要实现一个抽

象的联系实体来确保只发生有效的改变。但是如果你想要知道John和Mary Smith在1953年结婚并在1972年离婚，以及Mary在1975年再次结婚、但在1986年守寡，那么就需要一个抽象的联系实体来跟踪这些变化。

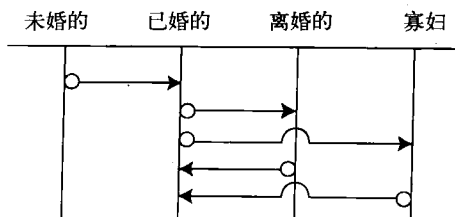


图3-5 状态转变图描绘了一个实体状态的合法变化，该情形是个人婚姻状态的变化

3.3 一对一联系

可能最简单的联系类型就是一对一联系。如果实体X的任何一个实例都只与实体Y的一个实例相关联，那么称这种联系为一对一的联系。大多数“IsA”类型的联系都是一对一的，但是在问题域中这样的联系并不多见。

需要特别注意的一件事是：当在实体间选择一个一对一联系时，需要确保该联系或者是永远正确，或者当它变化时不用在意过去的数值。例如，如果你在对一个写字楼里的办公空间进行建模，假定每个办公室只有一位工作人员，那么就在Office和Employee之间就存在一个一对一联系，如图3-6所示。

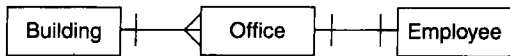


图3-6 在Office和Employee之间存在一个一对一联系

但是一名雇员和一个办公室的联系仅仅是在特定的时刻有效。过些时候，不同的雇员会被分配到同一个办公室。（该写字楼里的办公室也会变化，但这是不同的问题。）如果使用图3-6所示的一对一联系，你将有一个简单清晰的写字楼模型，但是没有办法确定空间占用的历史。

你可能不会在意。如果你在为邮件空间构建一个系统，你需要知道将Jane Doe今天的邮件寄向哪里，而不是三个月前的地址。但是如果你为一个财务经理设计一个系统，你就不能丢失这些历史信息——例如，人们可能要从系统中知道产权变化的频率是多少。

虽然一对一联系在现实世界中比较少，但是他们是十分常见的抽象结构，并且也是十分有用的。它们大多时候用在两种情形下：你希望（或需要）减少关系中的属性数目或者对子类实体进行建模。

如果使用Jet数据库引擎，那么每张表中都存在最多只能有255个字段的物理限制；但如果使用SQL Server表，那么每个表的平均字段数的限制是1024个。我怀疑——非常怀疑——任何一个数据模型会超过这些限制。但是我偶尔会看到一些系统，特别是用在科学和医学方面的系统，它们中的实体确实会有超过255个属性。在这些情况下，你别无选择，只能创建一个新表来存储属性的一些子集，并在新表和原表之间创建一个一对一的关系来控制新表。

如果对一个测试或调查问卷进行建模，那么问题域常常显式地要求扩展表大小的物理限

制。给定一个任意数量的问题的测试，你可能试图将每个人的回答建模为图3-7所示的形式。

这种结构很容易实现，特别是如果数据库仅仅被用于这种测试，并且只有这种测试。但这并不是一个好的解决方案，而且当你有数目不同的问题的测试时，这种结构就会失败。这里的答案（Answer）属性是一个重复组，因此该联系不是第一范式的。更好的建模方法如图3-8所示。

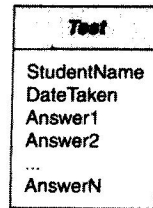


图3-7 这个结构有时候被用来对测试和调查问卷进行建模，但并不理想

子类实体

一对一联系的一种更有趣的用法是针对子类实体，它是从面向对象程序设计中借鉴的概念。为了发现子类实体的益处，让我们首先看一下传统的实现方式。在Microsoft Access中，Northwind样板数据库中的每个产品都属于某个产品类别，如图3-9所示。

为了报表的目的或者个人问题域的需求，可以建立一个类别（Categories）联系将产品分组。但是在这种设计中，你可能将一个产品仅仅作为一个产品来看待，而不是它所属的特定类别的一个实例。任何为Products定义的属性都被存储在所有产品中；不管它是何种类别。这与问题域不太符合——饮料（Beverages）和调味品（Condiment）本质上有不同的属性。

你可能试图将Northwind产品列表建模为如图3-10所示的形式。该模型允许我们存储每个特定产品类别的所有信息，因为每个关系都可以有不同的属性集合。这种模式的问题是它很难将产品仅作为一个产品对待。

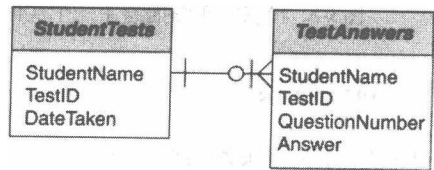


图3-8 虽然这种结构实现起来较难，但它更适合于对测试和调查问卷进行建模

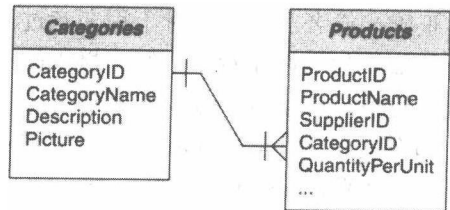


图3-9 Northwind数据库中的每个产品都属于某个产品类别

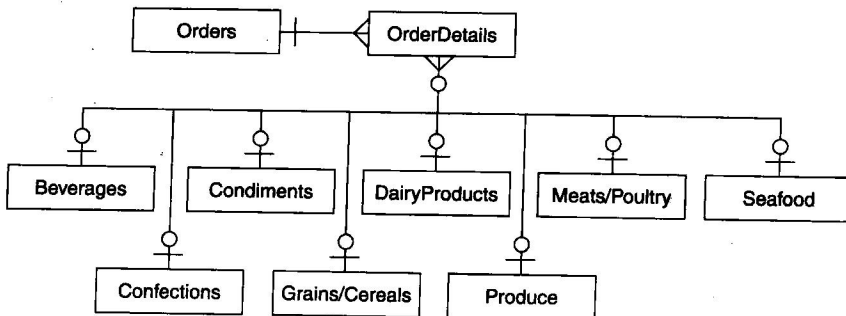


图3-10 这个模型允许得到特定产品类别的属性

例如，假设检查一个由用户输入的产品编码是否正确的过程：“如果该编码存在于X关系，或者Y关系，或者……”。这与第2章中提及的重复组一样会十分难看。同样地，如果你要将某

些属性只应用到一个产品类别上（比如，UnitsperPackage可能属于Beverages而不属于DairyProducts），而该特定产品的类别发生了改变，那么使用这种结构，可能就会涉及完整性问题。遇到这些情景你该如何处理呢？将原来的数值都丢弃吗？但如果该改变是意外的，用户又立刻将它改回来了呢？

子类产品实体提供了最好的解决方法。你可以为某些特定的产品类别获取具体信息，而不会失去在恰当的时候，简单将产品仅作为产品对待的能力，并且你可以延迟删除不再使用的信息，直到你十分确定它绝对不会再使用了。图3-11显示了一个使用子类实体开发的模型。

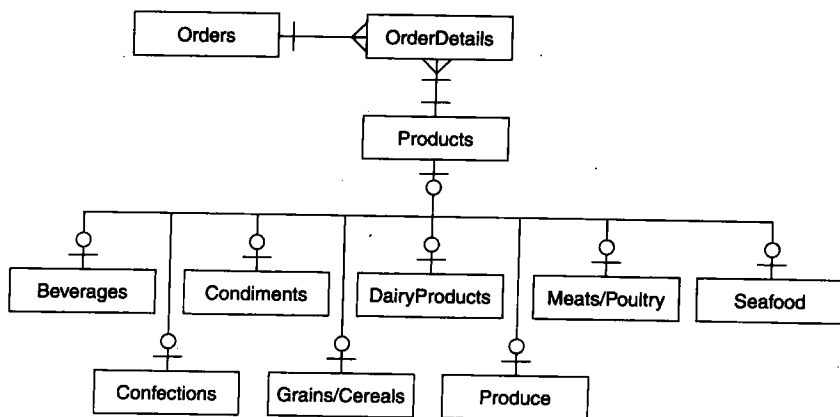


图3-11 该模型使用子类来提供图3-9和图3-10的功能

不得不承认，子类实体是对特定数据建模问题最完美的解决方法，但是它实现起来却十分困难。举个例子来说吧，一个包含产品细节信息的报表需要根据条件的处理过程以便仅仅显示当前子类的合适字段。这不是一个不可逾越的任务，但是它需要慎重地考虑。在大多数情况下，并不推荐你为了让程序员的工作变得轻松一些，而对数据建模做出让步。但是如果仅仅为了报表的目的而对实体分组和分类的话，通过增加子类从而增加系统的复杂性也是绝对没有必要的。这种情况下，图3-9的结构就足够胜任了。

在一对一联系中确定主关系和外部关系有时候比较困难。和通常一样，需要在数据模型的语义基础上作决定。如果你选择子类实体的结构，那么普通实体就应是主关系，而每一个子类应成为一个外部关系。

注意：在这种情况下，子类需要的外码通常就是子类的候选码。没有必要让子类使用它们自己的标识。

另一方面，如果使用一对一联系来避免字段限制，或者实体在问题域中本身就有一对一联系，那么主关系和外部关系的选择就要根据自己的理解了。必须在你对问题域的理解的基础上选择主关系。

在这种情况下，关系的可选性可以有一定的帮助。如果该关系只在一端是可选的（我从没见过在一个一对一联系中两端都是可选的），那么在可选端的关系就是外部关系。换句话说，如果实体中仅有一个实体是弱实体，其他都是常规实体，那么常规实体就是主关系，而弱实体是外部关系。

3.4 一对多联系

实体间最常见的联系类型是一对多联系，即一个实体的一个实例与另一个实体的零个、一个或者多个实例相关联。第2章讨论的大多数规范化技术都是针对有一对多联系的实体的。

一旦确定了一对多联系，它们就不会存在太多问题。但是，在指定联系的每一端的可选性时需要特别仔细。通常认为只有联系的多端才能是可选的，但并不总是如此。

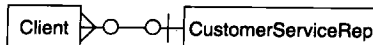


图3-12 这个联系在两端都是可选的

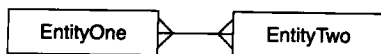
Client和CustomerServiceRep之间的联系在两个方向都是可选的。用自然语言来说，它可以被描述为：“CustomerServiceRep可以有一个或多个客户。如果指定了客户的CustomerServiceRep，则其必须出现在CustomerServiceRep关系中”。指定一对多联系中“一”端的可选性对于系统的实现和可用性都起到了重要作用。我们将在本章的后面详细讨论这些问题，但是这里需要理解的是关系理论并不要求一对多联系中的“一”端是被强制的。

在一对多联系中确定主关系和外部关系十分简单。处在关系中的“一”端的实体总是主关系；它的候选码被拷贝到处在“多”端的实体中，这个“多”端的实体就成为了外部关系。主关系中的候选码常常会成为多端的外部关系的候选码的一部分，但这些候选码本身不能唯一的标识外部关系的元组，它必须与其他一个或多个属性共同组成一个候选码。

3.5 多对多联系

多对多联系在现实世界中大量存在。学生学习多门课程；任何给定的课程都有多名学生参加。客户在多个商店买东西；一个商店有多名客户。但是在关系数据库中不能直接表达多对多联系，它们需要使用一个中间关系来建模，该中间关系与每个原参与者都存在一对多联系，如图3-13所示。这样的中间关系通常被称为连接表（junction table），既然讨论的是数据建模阶段，我们当然谈论的是关系而不是表格。

这个多对多联系：



被建模成如下形式：

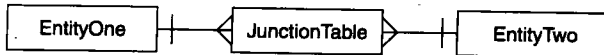


图3-13 被称作连接表的中间表被用来解决多对多联系问题

因为一个多对多联系被构建为两个一对多联系，那么确定主关系和外部关系就变得十分清楚了。正如我们已经看到的，一对多联系中的“一”端关系永远都是主关系。这就意味着每个原实体都将成为主关系，并且连接表将是外部关系，并且包含两端表中的候选码。

连接表通常都只包含两个原参与者的候选码，但是它们的确仅仅是之前讨论的一个抽象联系实体的一种特殊情况。例如，连接表解决了“Courses”和“Students”之间的多对多联系，当特定学生参加特定课程时，它可能包含了一个表明学期的属性。（并且，该学期属性可能参与到一个有一个学期实体的一对多联系中。这样的模式根据要求可以是简单的，或者是比较复杂的。）

3.6 一元联系

至此为止我们讨论的所有联系都是二元联系，它有两个参与者。一元联系只有一个参与者——关系与其自身相关联。一元联系中最经典的例子是Employee和Manager之间的关系。某个人是一名经理，在大多数情况下，他也是他所属经理的雇员。

一元联系可以是任何基数的。一对多一元联系被用来实现层次关系，比如暗含在Employee-Manager联系中的企业层次关系。多对多一元联系，和其他的二元关系类似，必须通过一个连接表来构建。一元联系在“一”端也可以是可选的，如图3-14所示。大多数企业的CEO都没有自己的上层经理。（股东不算在内，除非他们是该数据模型独立的一部分。）

一元联系使用和二元联系同样的方式建模——主关系的候选码被添加到外部关系中。唯一不同的是主关系和外部关系是相同的。因此，如果Employee表的候选码是EmployeeID，在EmployeeID域上声明，那么还需要在关系中添加一个称作ManagerID的属性，同样也是在EmployeeID域上声明，如图3-14所示。

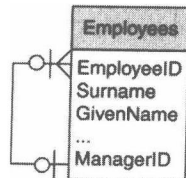


图3-14 当关系与其自身相关联时就存在一个一元联系

这样会存在一个十分有趣的问题：由于某种可能很明显的理由，一个表中不允许有两个同样名字的字段。（你不应当在一个表中有名字相同的两个字段，但是“名字”在抽象层面上是一个十分狡猾的概念。）因此，你必须在一元联系中重命名主码或者外码。大多数时候要重命名外码。在我们的例子中，你很可能称外码为“ManagerID”。

事实上，你可以在任何关系中重命名外码属性，并且这么做通常是有语义意义的。例如，Employee关系中的EmployeeID可以成为Customer关系中的CustomerServiceRepID。事实上开发环境会对组合表格做一些智能的推测，如果盲目的将一个关系中的字段名拷贝到另一个关系中，那么开发环境会建议不构建基于相同名字字段的表的组合。

3.7 三元联系

三元联系常常以X对Z做了Y的形式存在，并且和多对多联系一样，它们不能直接在关系数据库中建模。但是，与多对多联系不同的是，这里不存在某个变通的方法来对它们建模。

在图3-15中，我们可以看到被Vins et alcools Chevalier购买的Mozzarella di Giovanni同时被Formaggi Fortini s.r.l以及Forets d'erable提供，但是没有办法确定到底是哪家供应商提供了特定的奶酪配送给Vins et alcools Chevalier。在该数据模型中丢失了一个三元联系。供应商不会简单地只提供产品，他们还需要提供由特定客户购买的产品。

为了理解这个问题，首先在一个更典型的问题域中检查这种联系是十分必要的，如图3-16所示。

在图3-16中，每种产品仅由一个供应商提供，由此这个三元联系是可以维护的——如果你知道了某个产品，也就知道了是谁提供的。但是，在图3-17中，每种产品都由多个供应商提供，因此这个三元联系就被丢失了。

Customer ID	Company Name
VINET	Vins et alcools Chevalier

Customer	Order ID	Product Name
VINET	10274	Filet mignon
VINET	10274	Mozzarella di Giovanni
VINET	10295	Gnocchi di nonna Alice
VINET	10737	Konbu
VINET	10737	Jack's New England Clam Chowder
VINET	10739	Island of Sili
VINET	10739	Filo Mix

Product Name	Company Name
Mozzarella di Giovanni	Formaggi Fortini s.r.l.
Mozzarella di Giovanni	Forêts d'érables

图3-15 这些关系没能表明Vins et alcools Chevalier购买了哪个供应商的奶酪

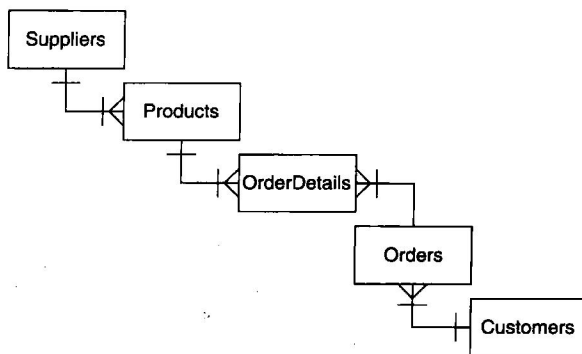


图3-16 这是一个参与订购的实体间典型的关系链

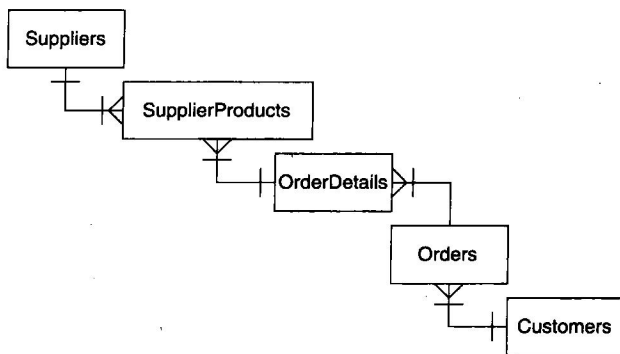


图3-17 在该模型中丢失了三元联系

解决这个问题的关键是检查一对多联系的方向。给定任何“多”端的实体，就可以确定相应的处于“一”端的实体。因此，给定一个如图3-16中具体的OrderDetails项，你可以确定它所属的Orders项，并且知道了Orders项，你就能确定Customers。当然，该工作过程在其他方向也是如此：知道了OrderDetails项，就可以确定产品进而确定供应商。

但是反之就不对了。如果已经在联系的“一”端确定了一个实体，那么你就不能在“多”

端选择一个单一实体。该问题如图3-17所示。知道了一个OrderDetails项，就可以确定一个产品，但是知道了产品，并不能确定它所连接的SupplierProducts实体。

一种更简易的考虑该问题的方式是，在一个关系链中，你不能多次将方向从一对多改变到多对一。图3-16中的关系链仅在一处改变了方向，即OrderDetails。而图3-17中的链改变了两次，即OrderDetails和SupplierProducts。

从链中消除Products实体的解决方法如图3-18所示。

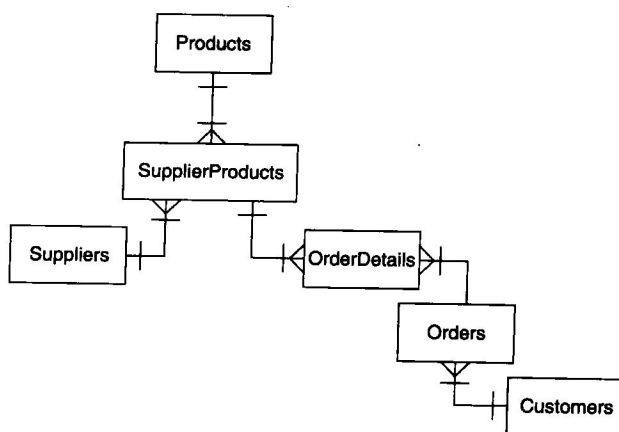


图3-18 该模型保持了三元联系

现在，链只在一处发生了改变，即OrderDetails，因此这个联系得到了维护。但是，要注意Products实体还没有被消除。不过对Products作订购会比对SupplierProducts要好很多，并且维护Products实体可以允许用户界面来调整它。

当然，你可能不在意问题域中的三元联系，或者在需要的时候存在其他的方法来跟踪该联系，比如包装上的一串数字。在这种情况下，的确没有必要对它建模。需要明白的是图3-18所示模型不一定就比图3-17所示的模型更好更正确，这一点十分重要。你必须选择最能反映你的问题域语义的模型。

3.8 已知基数的联系

偶尔在一个一对多联系中的“多”端的绝对最少或者最多的元组数目会预先得知。在学校的日子存在5个时期；在成人骨架中存在200根骨头；一名高尔夫球手在锦标赛中只能携带14根球棒。

通常对这些情况的建模是在关系的“一”端中包含每个元组的候选码，如图3-19所示。但是这种方法存在两个主要问题。第一，它是一个重复组；第二，它是不可靠的。

StudentNumber	GivenName	Surname	Period1	Period2	Period3
1	Nancy	Davolio	Biology	French	English
2	Andrew	Fuller	Physical Education	Biology	French
3	Janet	Levering	Physical Education	Biology	French

图3-19 像这样为已知基数建模是很诱人的，但很笨拙

属性的重复本质在图3-19中通过属性名字掩饰了，但是它们全都是定义在同一域上的——

ClassPeriod。每当你在相同的域上定义多个属性时，都能通过属性名称来对分类和类别进行伪装。

除了这个理论问题之外，这样的结构是不可靠的。例如，如果一个公司经理最多只能有五名雇员直接向他们汇报，这个策略对公司当然很好，但是制度不是现实。将制度设定到你的数据模型中，你将把它作为一个不可妥协的系统约束。我敢保证，你会发现在初始输入数据时，至少有一位经理拥有直属他的六名雇员。接着会发生什么呢？是不是某个人突然就会有一个新老板呢？或是该经理会被输入两次（说不定会付两次薪水）？又或者实现你设计的程序员在凌晨三点接到一个电话，述说对你的满腹抱怨呢？

类似这些基数上的限制必须作为系统的约束来实现，它们不应当被添加到关系本身的结构中。更进一步地，我们会在第19章讨论到，你应当在实现基数限制之前，更多更长远地考虑可用性的影响。

3.9 小结

在这一章中，我们细致地学习了实体之间的联系，我们介绍了每种类型的二元联系——一对一、一对多以及多对多——并且知道了如何在数据建模中表达它们：建立一个主关系，并将其候选码包含在其他联系参与者中，即外部关系中。除此之外，我们还学习了特殊的一元和三元联系，同样也知道了如何在数据建模中描述它们。

你已经知道了所有的数据模型的基本组件：实体、它们的属性，以及它们之间的联系。在第4章中，我们将转向数据完整性以及一些维护数据库一致性的机制。

第4章 数据完整性

对问题域中的实体以及它们之间的联系创建模型只是数据建模过程的一部分。此外，你还必须注意数据库系统将要使用的规则，它用来确保存储在其中的实际物理数据的正确性，即使不正确，至少也应当是看似真实的。换句话说，你必须对**数据完整性**建模。

应当意识到仅仅保证数据在字面上的正确性的是不够的，这一点很重要。举个例子来说，一个订购记录表明Mary Smith在1999年7月15日购买了17根钢锯。数据库系统能保证Mary Smith对系统来说是一名客户，而公司的确售卖了钢锯，并且该订购活动发生在1999年7月15日。它甚至能检查Mary Smith是否有足够的信用来购买17根钢锯。而它不能做的是检验Ms. Smith确实是订购了17根钢锯，而不是7根或者1根，或者17把螺丝刀。系统能够可能做到的最好方式是提示用户17根钢锯对于个人购买来说是一个大数目，提醒输入人员关注该结果。事实上，要让系统做到这一点，实现起来也许会十分复杂，很可能会大大提高成本预算。

我们的观点是系统在Mary Smith做出订购后，并不检查它所记录的信息，可以只检查她是否确实能够做这个订购。当然，所有保存记录的系统都能做到这点，如果除了应用一致性规则之外别无其他原因的话，一个设计良好的数据库系统一定会比普通的手工系统要出色。但是没有任何数据库系统，也没有任何数据库设计者可以保证数据库中的数据是正确的，所能保证的仅仅是它可能是正确的。它是通过实现为它定义的**完整性约束**来做到这一点的。

4.1 完整性约束

一些人将完整性约束看成是业务规则。其实，业务规则的概念要广泛得多，它包括所有系统中的约束，而不仅仅是对数据完整性的约束。特别地，系统安全性——它是用来定义用户在什么环境下可以做什么样的操作——就是系统管理的一部分，而不是数据完整性。但是特定的安全性是一种业务要求并且可以组成一个或多个业务规则。我们在第13章中讨论系统管理时，会看到数据库安全问题。

数据完整性可以在不同级别的粒度上实现。域、转换以及实体约束定义了维护每个关系的完整性的规则。参照完整性约束确保了维护关系之间的联系的重要性。数据库完整性约束将数据库控制为一个整体，并且事务完整性约束控制数据是在单个数据库中操作还是在多个数据库中操作。

4.1.1 域完整性

正如我们在第1章讨论的，一个域是一个给定属性的所有可能值的集合。域完整性约束——通常称为**域约束**——是用来定义这些合法值的规则。很可能需要定义多个域约束来完整地描述一个域。

域和数据类型是不同的，而依据物理数据类型来定义域是非常吸引人（也很普遍）的做法，但它可能事与愿违。其危险性在于你对某些值作了没必要的约束——例如，你选择一个Integer数据类型，只是考虑在实践中它已经足够大了，而不是因为255是该域所允许的最大数值。

然而，可以这么说，数据类型在数据模型中是一种比较便捷的实现方式，因此，出于这个原因，在确定系统中的域约束时，选择一个合适的逻辑数据类型通常是第一步要做的事情。所谓的**逻辑数据类型**，指的是“date”、“string”或“image”，不包括其他更具体的类型。Date类型很可能是该方法的最好例子。我不建议将域TransactionDate定义为“DateTime”，这个类型仅仅是一个物理表达。但是将它定义为“Date”则允许你将焦点集中在“业务开始日期和当前日期之间”，而忽略所有那些关于闰年的规则等。

一旦选择了一个逻辑数据类型，它可能恰当地缩小了数据定义的范围，比如表明了一个数字类型的精度和范围，或者字符串数值的最大长度。这非常接近于指定一个物理数据类型，但是你仍然工作在逻辑层面。显然，如果你把“一个不超过30个字符的字符串数值”十分简短的记作Char[30]，你也不会因此获得什么好处。但是你在数据模型中描述的越抽象，你在之后操作时将会获得的空间越大，并且也就越不太可能将意外的约束强加给系统。

域完整性需要考虑的下一个方面是一个域是否应当包含未知的或者不存在的值。如何处理这些值一直存在着争议，我们在讨论数据库系统设计的不同侧面时都会反复提及它们。现在，只需理解的是在未知值和不存在值之间存在差别，并且常常（虽然并不总是如此）可能在域中允许它们中之一或者两者都存在。

首先，“未知值”和“不存在的值”的不同性在逻辑层上并没有表现出太多问题。（并且请永远记住数据模型是一个逻辑结构。）我的父亲没有中间名字，我不知道我邻居的中间名。这些是不同的问题。这里存在一些不需要我们关心的实现问题，但是这种逻辑上的不同性是显而易见的。

第二点是当已经决定一个域中是否允许包含未知值和不存在的值之后，还需要确定它们是否能被系统所接受。回到TransactionDate的例子，一个事务的日期当然可能是未知的，但是如果它终究要发生，那么它必然在一个确定的时间点发生，因此它不可能不存在。换句话说，一定存在一个事物日期，我们可能只是尚不知道它而已。

现在很明显，对任何事情我们都可能不知道，因此任何值都可以是未知的。这并不是一个有用的区分方法。我们在这里真正要定义的不是一个值是否是未知的，而是是否应当存储一个拥有未知值属性的实体。有可能除了已知值之外，未知值是不值得存储的，或者我们在不知道值的情况下是不能标识一个实体的。不管是哪种情况，在将特定字段添加到数据库时，都要防止记录中包含未知值。

在域级别上，这个决定不总是正确的，但是既然这么做可以使得工作稍微简单一些，那它总是值得考虑的。在某种程度上，做决定的依据应当是域的性质。举个例子，比如你已经定义了一个Name域并且声明了GivenName、MiddleName、Surname属性，以及与之对应的CompanyName属性。你可能也为这些属性分别定义了各自的域，但使用一个更通用的域可能会更好，因为这么做可以允许使用可重用的规则（在这个例子中，很可能存在很多这样的情况）。但是，在这种情况下，你不能决定在域的级别上是否可以接受空值和未知值，必须在实体级别定义这些特征。

域完整性的最后一个方面是要尽可能具体地定义由一个域所描述的值的集合。例如，我们的TransactionDate域不仅仅是所有日期的集合，而且它是从公司开始交易的日期至当前日期之间的所有日期的集合。它可能需要进一步限制，来消除周日、公共假日以及任何公司不做交易的日期，你需要对此格外细致。即使在业务的正常休息日，也可以知道雇员是否在办

公室，并且当雇员在办公室时也绝对没有理由阻止雇员接听电话并接受订购。

有时候，最简单的描述一个域的方式是列举域的所有值。Weekends域完全可以由集合{“Saturday”，“Sunday”}来描述。有时列举一个或多个规则来约束域成员会更加容易一些，比如我们为TransactionDate定义的规则。这两种技术都完全可以接受，虽然一个特定的设计方法可能规定了一个特定的文档约束形式。重要的是这些约束要尽可能的细致和完整。

4.1.2 转换完整性

转换完整性约束定义了一个元组应该遵循的状态。例如，图4-1所示的状态转换图显示了一个订购过程可以经过的状态。

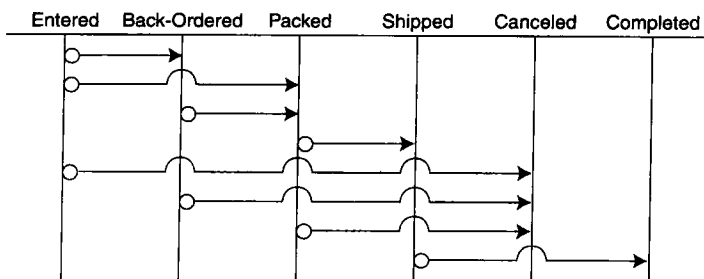


图4-1 该图显示了一个订购可以经过的状态

比如，你可以使用转换完整性约束来确保一个给定的订购状态不可能不通过其他中间状态从“Entered”变化到“Completed”，或者可以阻止一个取消的订购再改变其状态。

实体的状态通常由一个单独的属性来控制。在这种情况下，转换完整性可以被看成是一种特殊类型的域完整性。但是，有时候有效的转换需要多个属性甚至是多个关系来控制。因为转换约束可以在任何粒度的级别上存在，在数据模型的准备阶段，将它们单独作为一个约束类型来看待是十分有用的。

例如，如果一个客户的信用限制超过了一个指定的值，或者他与公司的业务往来至少有一年的时间，那么就可能允许该客户的状态从“Normal”改变为“Preferred”。信用限制要求很可能被Customers关系的某个属性记录，但是该客户与公司的业务往来时间可能非显式地存在某个地方。也许有必要通过该用户在Orders关系中的最早记录来计算业务往来时间。

4.1.3 实体完整性

实体约束确保由系统所建模的实体的完整性。最简单的，主码的存在就是一个实体约束，它强迫服从这样一个规则：“每个实体必须能被唯一确定”。

在某种意义上，这是实体完整性约束，所有其他的都是技术上的实体级别的完整性约束。定义在实体级别的约束可以控制单个属性、多个属性或者整个关系。

单个属性的完整性建模首要的是针对一个特定的域定义属性。关系中的属性继承了定义在其域上的完整性约束。在实体级别上，这些继承的约束适当地可以制定得比较严格一些，而绝不能放松它们。另一种考虑的方式是实体约束可以指定一个域约束的子集，而不是一个超集。例如，针对TransactionDate域定义的OrderDate属性可能指定日期必须是在当前年，而

TransactionDate域允许从业务开始日期至当前日期的所有日期。但是一个实体约束不能允许OrderDate包含将来的日期，因为属性的域限制了这一点。

同样地，一个针对Name域定义的CompanyName属性可能禁止空值，即使Name域允许空值存在。这是比域中指定的值的范围更小、要求更严的可允许的值。

注意：设计者常常在实体级别指定空值和未知值的有效性，而不是在域级别中指定。事实上，一些设计者可能会认为这些约束应当仅应用在实体级别。这样的论断有一定的合理性，但是我建议将域定义得越广泛越好。当然，在域级别考虑空值和未知值并没有什么危害，并且可能使描述（和实现）的过程更简单。

除了可以缩小单个属性的值范围外，实体完整性还可以影响多个属性。确保ShippingDate不得在OrderDate之前的要求就是一个这类约束的例子。但是，实体约束不能参照其他的关系。例如，定义一个实体约束使得一个客户的DiscountRate（Customer关系中的一个属性）是基于该客户的TotalSales（它基于OrderItems关系的多条记录），这样的约束是不合适的。基于多个关系的约束是数据库级别的约束，我们将在本章的后面讨论它。

必须小心多属性的约束，它们常常表明你的数据模型没有被完全规范化。如果你基于其他的属性来限制或计算另一个属性的值，它可能还行得通。一个实体约束是这样的：“除非Customer记录至少有一年历史，否则Status是不允许变为‘Preferred’的”，这是没有问题的。但是如果一个属性的值决定了另一个属性的值——例如，“如果该客户的记录已超过一年的历史，那么Status = ‘Preferred’”——这样就存在一个函数依赖，并且违反了第三范式的要求。

4.1.4 参照完整性

在第3章中，我们介绍了通过对关系进行分解以减少冗余的方法，以及通过外码来实现这些关系之间的连接。如果这些连接不存在，那么该系统至少是不可靠的，更糟糕的情况是系统根本不可用。**参照完整性**维护并保证了这些连接。

实际只存在一种参照完整性约束：外码不能孤立。换句话说，外码表中不允许存在与主表的记录不相匹配的外码记录。包含外码的元组在主关系中没有对应的候选码的现象称作**孤立实体**。有四种可能创建孤立实体的方式：

1. 一个元组被添加到外部表中，而在主表中没有与之匹配的候选码。
2. 外码改变了，使其值不在主表中了。
3. 主表中的候选码的值改变了。
4. 被引用的主表中的记录被删除了。

要维护一个联系的完整性，就要处理以上所有这四种情况。第一种情形，添加了一个不匹配的外码，通常是很容易阻止的。但是要注意的是未知值和不存在的值是不算在内的——那是另外的规则。如果该联系被声明为可选的，那么任意数量的未知值和不存在值都可以输入，而不会损害参照完整性。

第二和第三种情况——改变了参照表中候选码的值——造成的孤立实体不会经常发生。事实上，我们强烈推荐在任何时候都禁止候选码和外码的变更。（顺便说一下，这也是一个实体约束：“候选码不允许改变”。）

如果你的模型允许改变候选码，那么必须确保相应的外码也做必要的改变。这被称作**级联**

更新。Microsoft Jet和Microsoft SQL Server都提供了这样的机制以便十分容易地实现级联更新。

如果你的模型允许改变外码值，结果其实是允许重新指定实体，那么你必须确保新的值是有效的。同样的，Microsoft Jet和Microsoft SQL Server都允许较容易的实现这样的约束，即使它还没有一个特定的名称。

最后一种造成孤立外码的情况是删除包含主实体的记录。例如，如果某人删除了一个客户记录，那么该客户的订购信息将如何处理呢？如同改变候选码一样，如果它们被外码表引用的话，你可以简单地禁止删除主表中的元组。如果这对你的系统是合理限制的话，那么该解决方案当然是最清楚简单的。如果不是，Jet数据库引擎和SQL Server都提供了一种简单的级联操作方式，被称作**级联删除**。

但是在这种情况下，你也有第三种选择，不过由于它不能自动实现，所以实施起来比较难一点。你可以尝试重新指定要引用的记录。这种方法并不总是合适的，但它有时候是必要的。例如，如果CustomerA替换CustomerB，那么就需要删除CustomerB，并把所有CutomerB的订购重新指定给CustomerA。

一种特殊类型的参照完整性约束就是第3章讨论的最大基数问题。在这种数据模型中，诸如“经理最多只允许有五名直属于他们的雇员”这样的规则就应当被定义为参照约束。

4.1.5 数据库完整性

最常见的完整性约束的形式就是**数据库约束**。数据库约束通常涉及多个表：“销售给状态为‘Government’的客户的的所有产品一定有超过一个的供应商。”大多数的数据库约束都是这种形式，比如像这个例子，可能还需要对多个关系重新进行计算。

你必须十分注意不要将一个数据库约束和一个具体的工作过程相混淆。一个**工作过程**是由数据库所作的操作，比如添加一条订购信息，而一个完整性约束是关于数据库内容的规则。这些定义了由数据库实施的的任务的规则是工作过程的一部分，而不是数据库约束。我们将会在第11章看到，工作过程对数据模型有着重要影响，但是它们不应该是由其中的一部分。

是否应当将一个给定的业务规则作为一个完整性约束或者一个工作过程（或者完全是别的什么），并不总是很清楚的。这之间的区别也许并不十分重要。只要它们在最合适的地方实施，都会有一样的效果。如果将一条规则作为数据库约束来描述是一个十分简单的过程的话，那么就这样实施好了。如果这样实施比较困难的话（通常是这样的，即使该规则显然就是一个完整性约束），就将其移至中间层或者前端来实现，在那里它能由程序来实现。

4.1.6 事务完整性

最后一种数据库完整性是**事务完整性**。事务完整性约束管理的是数据库操作的方式。同其他的约束不同，事务约束是用程序实现的，因此它本质上不是数据模型的一部分。

事务与工作过程紧密相关。事实上，这两个概念是正交的，因为一个给定的工作过程很可能由一个或多个事务组成，反之亦然。这种说法并不完全正确，但是对理解工作过程是一种抽象结构（“增加一次订购”）而一个事务是一个物理概念（“更新OrderDetail表”）是很有用的。

一个事务通常被定义为一个“逻辑工作单元”，这其实是一个毫无帮助的修辞语。本质上来讲，一个事务是一个动作组，所有这些动作要么都完成，要么都不做。数据库必须在事务

开始前或者事务完成后符合所有已定义的完整性约束，但是在事务过程中，很可能临时的与一个或多个约束相冲突。

最经典的事务的例子就是将钱从一个银行账户转到另一个银行账户。如果资金从A账户借出，但是系统没能将其注入到B账户中，那么钱就流失了。显然，如果第二个命令失败了，就应当取消第一个命令。用数据库的术语来说，这叫做“回滚”。事务可能会涉及多个记录，多个关系，甚至多个数据库。

为了精确起见，所有针对数据库的操作都是事务。甚至更新一条单个记录也是一个事务。不幸的是，这些低级别的事务都是由数据库引擎来实现的，对用户来说是透明的，你很可能就忽略了这个级别上的细节。

Jet数据库引擎和SQL Server都提供了维护事务完整性的方式——BEGIN TRANSACTION、COMMIT TRANSACTION，以及ROLLBACK TRANSACTION语句。正如所期望的一样，SQL Server的实现更加强大，并且能更好地从软硬件的失败中恢复过来。但这些实现问题超出了本书的范围。从设计的观点，重要的是获取和指定事务的范围，那些所谓的“逻辑工作单元”。

4.2 实现数据完整性

自此，我们都是在概念数据模型中的抽象层面关注问题域。在本节，我们将介绍一些在问题域中创建物理模型所涉及的问题：数据库模式。从一个层面转到另一个层面，主要是术语发生了变化，除了数据完整性的问题之外，关系转变为表，属性转变为字段。不过，这些映像从来就不如人们所期望的那样清晰。

4.2.1 未知值和不存在的值

在本章的前面，我们比较简单的说明了域和属性都应当检查是否允许空值或者未知值。但是还没有讨论如何实现这些约束。既然我们转向了数据库模式，那么就不能再逃避这个实现问题了。

所谓的“缺失信息问题”在关系模型被首次提出时就得到了认同。人们如何能分辨出任意给定的信息是缺失的（客户不是没有姓氏，而是我们不知道而已）或者是不存在的（客户没有中间名）？大多数关系数据库，包括Microsoft Jet数据库和SQL Server数据库，都使用Null作为对缺失值和不存在值的处理方式。

将Null称作是该问题的解决方法有点言过其实，因为它依然存在很多问题。一些数据库专家完全反对Null。C.J.Date声称它们“破坏了模型”，而且我已经不记得有多少次听到将它们称作是“恶魔”。任何关于处理Null的复杂性的评论以及关于后悔已经使用它们的说法，最终的结果都是这样的：“好了，你不应该使用它们，它们是有害的”。

作为一种替代方案，“Nulls是恶魔”学派的人建议使用一个恰当域的具体值来表明未知或不存在的值，或者同时描述这两种情况。我们称之为常规值方法。这种常规值的方法存在几个问题。首先，在很多实例中所选择的值都是常规的。一个日期9/9/1999并不意味着该日期是未知的，我们仅仅认同它是这样的意思。并且如果它是类似“只有别人告知时你才能知道”的情形之一，那么它会给系统和用户带来没必要的负担。

我不认为这是一个替代Null的更好的方法。其实，Null也是一个常规值，但它不会与其他

任何值相混淆，并且它还有其他优点——被关系模型和大多数数据库引擎所支持。

第二，常规值的另一个问题是其对参照完整性的影响，但在我看来这一点是不能容忍的。举个例子来说，在Customer和Customer Service Representative（客户服务代表，CSR）之间存在一个可选性联系，比如如果指定了一条记录，那么相应的CSR就应当在CSR表中列出。常规值方法要求被添加到CSR表中的新记录必须与所选择的常规值相匹配，以表明尚未分配CSR，如图4-2所示。

CustomerID	CompanyName	CSR
ANATR	Ana Trujillo Emparedados y helados	Andrew Fuller
ANTON	Antonio Moreno Taquería	Anne Dodsworth
AROUT	Around the Horn	Steven Buchanan
BERGS	Berglunds snabbköp	Margaret Peacock
BLAUS	Blauer See Delikatessen	UNASSIGNED
BLOMP	Blondel père et fils	Steven Buchanan
BOLID	Bólido Comidas preparadas	Nancy Davolio
BONAP	Bon app'	UNASSIGNED
BOTTM	Bottom-Dollar Markets	Janet Leverling
BSBEV	B's Beverages	Margaret Peacock
CACTU	Cactus Comidas para llevar	Michael Suyama
CENTC	Centro comercial Moctezuma	UNASSIGNED

EmployeeID	Surname	GivenName
1	Davolio	Nancy
2	Fuller	Andrew
3	Leverling	Janet
4	Peacock	Margaret
5	Buchanan	Steven
6	Suyama	Michael
7	King	Robert
8	Callahan	Laura
9	Dodsworth	Anne
10	UNASSIGNED	

图4-2 常规值要求添加“虚构”的记录来维护参照完整性

现在，公司雇佣了多少CSR呢？应当是比表中所列出的CSR数目少一个，因为其中有一个是虚构记录。真糟糕！每位CSR平均有多少位客户呢？Customer表中的记录数减去与“UNASSIGNED”CSR匹配的记录数，然后除以CSR表中的记录数减一。更加糟糕了！

常规值在产生报表时是十分有用的。例如，你可能想要将Null值替换为“Unknown”，将空值替换为“Not Applicable”。但是这与在数据库中存储这些常规值是完全不同的命题，在数据库中，我们已经看到，它们将干扰数据操作。

也许Null是恶魔，并且它也不太好看，但是它是我们拥有的最好地处理未知值和不存在的值的方法。仔细想想该问题，如果替代方法合适就可以使用替代方法，但在替代方案不合适时，就应当允许使用Null。

Null所带来的一个问题是，除了域被声明为字符串或者文本数据类型，它们都要强制执行双重职责。声明为日期时间数据类型的字段只能接受日期和Null。如果相应的字段被定义为同时允许未知值和不存在的值，并且都用Null来表示，那么就没有办法决定在一个具体字段中的Null表示的是“未知”还是“不存在”。这个问题对字符串和文本数据类型就不存在，因为你可以使用空的、零长度的字符串来表示空值，让Null来表示未知值。

在实际中，这个问题并不像人们想像的那样频繁出现。极少数非文本域允许有不存在的值，

所以在这些域中，Null总是被理解为未知。对于那些允许接受不存在值的域，通常选择一个替代的方式来描述它。注意，我在这里建议的是一个实际的值，而不是一个常规值。例如，即使一个Product关系有一个Weight属性、一个Service Call属性，那些显然没有重量的字段可以使用零值。（零对于很多数字字段中描述空的情况都是一个很好对选择，但不是全部适用。）

有关Null的第二个问题，也是更严重的一个问题，它们使得数据操作复杂化。逻辑比较变得更复杂，并且会引起许多繁琐的问题。我们将在第5章细致地讨论这一点。

我们不会轻视Null值，当存在合理的替代方法时，建议使用替代方法。但是我們也在其他地方说过，不要为了减轻程序员的工作而损害了数据模型。仔细考虑，但是如果系统要求Null值，那么就使用它们好了。

4.2.2 冲突响应

当定义数据库模式时，不仅要定义怎样最大限度的有效实现一个给定的完整性约束，并且还要决定如果违反了该约束，数据库引擎该如何应对。当然，大多数情况下，数据库会简单的拒绝错误的命令，并用某个合适的方法返回一个错误信息。但是，有时候，数据库可以做出正确的响应以使该请求成为可以接受的。这样的例子有：给那些不允许空值的属性设定一个默认值，或者实施一个级联更新和级联删除来保护参照完整性。对于冲突响应的内容我们将在本书的第III篇详细讨论。

4.2.3 声明的和过程的完整性

关系数据库引擎以两种方式提供完整性：声明的和过程的。**声明完整性**是将完整性显式地定义为（声明为）数据库模式的一部分。Jet数据库引擎和SQL Server都提供了某些声明完整性的支持。在实现数据完整性时通常更倾向于使用声明完整性。你应该尽可能的使用声明完整性。

SQL Server通过**触发器**程序的方式实现了**过程完整性**，当插入、更新或者删除一条记录时，它就会执行（“触发”）。Jet数据库引擎不支持过程完整性。当使用声明完整性不能实现某个完整性约束时，并且系统使用的是Jet引擎，那么该约束必须在前端实现。

4.2.4 域完整性

SQL Server通过用户定义数据类型（UDDT）提供了有限的几种对域的支持。被定义为用户定义数据类型的字段将继承数据类型的声明和为该用户数据类型所定义的域约束。

同样重要的是，SQL Server禁止在两个声明中为不同的用户定义的数据类型的字段间做比较，即使该问题中的用户定义的类型是基于相同的系统数据类型。例如，即使CityName域和CompanyName域都被定义为char(30)，SQL Server也会拒绝这样的表达式：CityName = CompanyName。虽然这可以通过使用转换函数CityName = CONVERT(char(30), CompanyName)，但是该限制会迫使你在比较针对不同域声明的字段之前好好考虑一下。既然这样的比较没有任何意义，那么这种限制就是一件好事。

创建UDDTs可以通过SQL Server企业管理器或者系统存储过程sp_addtype实现。不管是哪种方法，定义UDDT时，都需要指定一个名字或者一个数据类型，并指定是否允许接受Null值。一旦创建好了一个UDDT，就可以为它定义默认值和有效性规则。一条SQL Server规则是一个

逻辑表达式，它为该UDDT（或者是一个字段，如果它是绑定到一个字段上而不是UDDT）定义了可接受的值。一个默认值是很简单的，它会在一个字段可能为Null时被插入到系统中，因为用户没有提供一个值。

给一个UDDT绑定一个规则或者默认值的过程要分为两步。第一，必须创建该规则和默认值，然后再将它们绑定到UDDT（或者字段）上。关于这个两步的过程有一个这样的论断：“它不是一个缺陷，而是一个特征”。因为一旦定义了规则或者默认值，这些规则或者默认值就可以在任何其他的地方重用。不过以我的经验，这一点很多余，因为这些对象很少会重用。当定义一张表时，SQL Server提供了直接声明默认值和CHECK约束的功能，并将它们作为表定义的一部分。（CHECK约束与规则类似，但是更强大。）不幸的是，这一步声明对于用户定义的数据类型是不可用的，它必须使用刚才那种“先创建，再绑定”的方法。十分衷心地希望微软会在将来的SQL Server版本中为UDDT添加CHECK约束和默认值声明的支持。

第二种实现域完整性的方法是使用查询表。这个技术在Microsoft Jet和SQL Server中都可以使用。作为一个例子，让我们看看USStates属性。现在，在理论上，你可以创建罗列全部50个州的一条规则。而实际上，这将是一个很痛苦的过程，特别是使用Jet数据库引擎时，因为你不得不为每一个针对该域声明的字段都重复输入一遍这些州。创建一个USStates查询表会简单很多，并可使用参照完整性来确保该字段的值被限制在查询表中所存储的值的范围内。

4.2.5 实体完整性

在数据库模式中，实体约束可以控制单个字段、多个字段或者整个表。Jet数据库引擎和SQL Server都提供了在实体级别上确保完整性的机制。很自然的，SQL Server提供了一套更丰富的功能，但这个差距并不像想像的那么大。

在单个字段的级别，大多数基础的完整性约束当然是数据类型。Jet数据库引擎和SQL Server都提供了丰富的数据库类型，如表4-1所示。

表4-1 Microsoft Jet和SQL Server支持的物理数据类型

逻辑数据类型	SQL Server数据类型	Microsoft Jet数据类型	值 域	存储空间
整型	Int	Long integer	-2 147 483 648~2 147 483 647	4字节
	Smallint		-32 768~32 767	2字节
	Tinyint		0~255	1字节
集合小数 (准确数)	Decimal	Number (类型可变)	$-10^{38} - 1 \sim 10^{38} - 1$	2~17字节
浮点数 (近似数)	Float(15位精度)	Double	数字部分大约: $-1.79E^{308} \sim 1.79E^{308}$ 正数范围: $2.23E^{-308} \sim 1.79E^{308}$ 负数范围: $-2.23E^{-308} \sim -1.79E^{308}$ 数字部分: $-3.40E^{38} \sim 3.40E^{38}$	8字节
	Real	Single	正数范围: $1.18E^{-38} \sim 3.40E^{38}$ 负数范围: $-1.18E^{-38} \sim -3.40E^{38}$	4字节
字符 (固定长度)	Char	N/A	在Jet中最大到255个字符，在SQL Server 7.0中是8000个字符（以前版本中是255）	每个声明的字符占1个字节

(续)

逻辑数据类型	SQL Server数据类型	Microsoft Jet数据类型	值 域	存储空间
字符(可变长)	Varchar	Text	在Jet中最大到255个字符, 在SQL Server 7.0中是8000个字符(以前版本中是255)	每个存储的字符占1个字节
货币	Money	Currency	数据精度到小数点4位, -922 337 208 685 477.5808~922 337 208 685 477.5807 数据精度到小数点4位, -214 748.3648~214 748.3647	8字节
	Smallmoney	N/A		4字节
日期和时间	Datetime	Date/Time	在SQL Server中是从1753年1月1日到9999年12月31日, 在Jet中是从100年1月1日到9999年12月31日	8字节
	Smalldatetime	N/A	从1900年1月1日到2079年6月6日	4字节
二进制(固定长度)	Binary	N/A	最多8000字节	声明的字节数加上4
二进制(可变长度)	Varbinary	(只对连接的表支持)	最多8000字节	存储的字节数加上4
大文本/二进制大对象(BLOB)	Text	Memo	在SQL Server中字符数据最多到2 GB, 在Microsoft Jet数据库中是1GB	存储的字节数加上16
	Image	OLE Object	在SQL Server二进制数据最多到2 GB, 在Microsoft Jet数据库中是1GB	存储的字节数加上16
布尔	Bit	Yes/No	0或1	1字节, 但在SQL Server中, 一个表中的bit数据被组合在一起, 因此, 少于等于8个的bit列将共同占用1个字节

正如我们在之前的章节中看到的, SQL Server也允许字段声明为UDDTs。一个UDDT字段继承了空值、默认值和为该类型定义的规则。从逻辑上讲, 该字段定义应该仅仅是缩小了用户定义的数据类型的约束, 但是事实上SQL Server会在字段描述中简单地替换UDDT的定义。因此, 即便一个UDDT已声明不允许Null值, 但仍旧可能允许该字段接受Null值。(甚至这么多年以来, 我都不能确定SQL Server是否缺乏精确性, 或者是我固步自封了。)

SQL Server和Jet数据库引擎都提供控制一个字段是否允许包含Null值的功能。当在SQL Server中定义一个列时, 用户可以简单地指定NULL或NOT NULL, 或者在企业管理器中点击合适的框。

Jet数据库引擎中与Null标志等同的是Required字段。除此之外, Jet数据库引擎还提供了AllowZeroLength标识, 它决定了在Text和Memo字段中是否允许空字符串(“”)存在。该约

束在SQL Server中是使用一个CHECK约束来实现的。

在Jet数据库引擎中,当定义字段的默认值时,简单地设定相应的属性即可。在SQL Server中,可以在创建该字段时设置Default属性,或者可以为该字段绑定一个系统的默认值,就像在UDDT里描述的一样。将默认值的声明作为表定义的一部分当然是很清楚的,如果你没有(或者不能)在域级别声明默认值,那么我十分推荐这种选择。

最后,Jet数据库引擎和SQL Server都允许建立特定的实体约束。Jet数据库引擎提供了两个字段属性:ValidationRule(有效性规则)和ValidationText(有效性文本)。SQL Server允许在定义字段时同时定义CHECK约束,或者在字段声明后绑定系统规则。CHECK约束是相对更好的方法。

粗略一看,Jet数据库引擎的有效性规则和SQL Server的CHECK约束似乎是相同的,但是它们之间有一些重要的差别。二者都是采用一个逻辑表达式的形式,并且它们都不允许参照其他的表或者列。但是,Jet数据库引擎的有效性规则等于True时,才能表明该值是可以接受的,而SQL Server的CHECK约束要求不等于False。这里有一个微妙点:对于CHECK约束来说,True和Null值都是可以接受的值;而对于有效性规则来说,只有True值是可以接受的。

除此之外,一个SQL Server字段可以定义多个CHECK约束。事实上,一个SQL Server字段可以有一条规则或者任意数目的CHECK约束,而Jet数据库引擎的一个字段只能有一个ValidationRule属性。另外,Jet数据库引擎的ValidationText属性的设置能返回给前端一个错误信息。Microsoft Access将文本显示在一个消息框里,在Visual Basic和其他的编程环境中,可以以错误集合的文本形式得到这些信息。

在一个表中参照多个字段的实体约束在Jet数据库引擎中是作为表的有效性规则来实现的,在SQL Server中是以表级CHECK约束的形式实现的。虽然在不同的地方声明,但这些表一级的约束功能与它们对应的字段一级的约束完全一样。

最基本的实体完整性约束就是要求实体中的每一个实例都是唯一确定的。记住这是实体完整性的规则;所有其他的则更多是指实体级上的完整性约束。Jet数据库引擎和SQL Server都以基本相同的方式支持唯一值约束,但对唯一值约束的支持看上去是很不相同的。这两个引擎都是使用索引来实现该约束,但是SQL Server对用户隐藏了这一点。用户到底是显式的创建了一个索引(Jet数据库引擎)还是声明了一个约束(SQL Server)很大程度上是机器的细节问题。

Jet数据库引擎和SQL Server都支持多个字段的集合是唯一的,同时也都支持定义一个或多个字段为主码,这其中就隐含了唯一性。虽然主码可以由多个字段组成,但一张表只能有一个主码。不过,一个表里可以有多个唯一值约束。

唯一值约束和主码之间还有另外一个重要的区别:唯一索引可以包含Null值,而主码不能。在唯一索引中对待Null的问题上,两个引擎有一些差别。Jet数据库引擎提供了一个属性:IgnoreNulls,它能阻止将包含Null值的记录添加到索引中。这些记录可以被添加到表中,但不能被添加到索引里。但SQL Server没有提供这样的功能。

除此之外,SQL Server在索引中只允许一个记录包含NULL。这在逻辑上是不正确的。它将有NULL值的记录看成是相同的,但显然它们是不同的。一个NULL值不等于任何其他东西,包括其他的NULL。

有趣的是,Jet数据库引擎和SQL Server都不要求在一张表中一定要定义主码或者必须有

唯一值约束。换句话说，创建一个非关系的表是可以的，因为关系中的元组必须是唯一确定的，而表中的记录却不需要这个限制。为什么这两个系统会允许这样做已经超出了我们要讨论的范围，但是我想即使你不需要它，了解这种可能性总是好的。

SQL Server为实现实体级上的完整性还提供了一种过程性的机制，而Jet数据库引擎则没有提供。触发器是一段代码，当一个特定的事件发生时它会自动执行。在SQL Server的目前的版本中，触发器中的代码必须是Transaction SQL。在被命名为“Yukon”的下一个版本中，允许使用任何.NET语言来编写触发器。可以为每个INSERT、UPDATE、DELETE以及新的INSTEAD OF事件定义多个触发器，并且一个给定的触发器可以用于多个事件。

4.2.6 参照完整性

Jet数据库引擎和SQL Server在支持实体完整性上本质是相同的，但它们在支持参照完整性方面的实现形式却不同。SQL Server允许外码约束作为表定义的一部分来声明。一个外码约束建立了对另一个表——主表的候选码的引用，一旦建立了该引用，SQL Server就能通过拒绝插入任何在主表中没有匹配的记录，来防止创建孤立记录。在外码列中没有禁止Null值的，如果该列也是该表的主码（通常是这样的）的一部分的话，就可以阻止Null值了。如果主表中的记录有与其匹配的外码值，则SQL Server还能禁止在主表中删除这些记录。

Jet数据库引擎通过数据库中的一个Relation对象来支持参照完整性约束。微软的术语在这里就显得很不恰当了——Jet数据库引擎中的Relation对象是对两个实体间联系的一种物理表达。不要将Relation对象和在数据模型中定义的逻辑关联搞混淆了。

创建Relation对象的最简单的方法是利用Microsoft Access用户界面（在Tool菜单中使用Relationships命令），但是它们也能通过编码创建。Data Access Object (DAO) 中的Relation对象的Table和ForeignTable属性定义了参与在联系中的两张表，而Fields集合定义了在每个表中的连接字段。

Jet数据库引擎是由该关联的Attributes属性来维护关系中的参照完整性的，如表4-2所示。

表4-2 在Jet数据库引擎中定义关系时使用的常量

属 性 常 量	说 明
dbRelationUnique	联系是一对一的
dbRelationDontEnforce	联系不是强制的（没有参照完整性）
dbRelationInherited	联系存在于一个包含两个连接表的非当前数据库中
dbRelationUpdateCascade	级联更新
dbRelationDeleteCascade	级联删除

注意属性标识dbRelationUpdateCascade和dbRelationDeleteCascade。如果设定了该更新标识，并且如果某个引用字段发生了变化，那么Jet数据库引擎将自动更新外码表中匹配的字段。同样的，删除标识将导致自动删除外码表中相匹配的记录。SQL Server 2000引入了与此对应的自动标识，不过更复杂的级联操作可以使用触发器简单的实现。

4.2.7 其他类型的完整性

在数据模型中，我们定义了三种附加的完整性约束：数据库约束、转换约束以及事物约

束。某些转换约束可以简单地通过声明有效性规则来实现。但是，大多数的转换约束，还有数据库和事物约束都需要通过程序实现。对于SQL Server来说，这意味着使用触发器。由于Jet数据库引擎没有触发器，这些约束只能在前端实现。

4.3 小结

在本章中，我们探讨了对数据完整性的建模和实现。存在三种完整性约束——域、转换和实体——来控制每个关系，而参照完整性约束维护的是关系之间的联系。最后，数据库约束和事物约束是将数据库作为一个整体来控制。

在数据库模式中是综合使用声明和程序的完整性来实现数据完整性控制的。声明的完整性就是显式的将声明作为模式的一部分，这种方法是被经常使用的实现方法。但是，不是所有的约束都能使用声明的完整性来实现的，所以在它们实现不了时，就要使用过程的完整性了。

在第5章，我们将学习关系代数以及在数据库的关系上可以实施的操作。

第5章 关系代数

在前面的章节中，我们已经介绍了如何定义一种特殊类型的关系，也叫做**基本关系**（base relation），它在数据库中将赋予一个物理描述。关系模型还支持创建多种类型的派生关系。**派生关系**（derived relation）是根据其他关系而不是根据属性来定义的。这些命名的关系可以是基本关系或者其他派生关系的任意组合。

在数据库模式中，一个基本关系可以由一张表来表达。派生关系在Microsoft SQL Server中可以用**视图**来表达，在Microsoft Jet数据库引擎中用**查询**来表达。为了语言上的简单明了，我将使用术语“视图”，因为它是标准的关系术语，这样可以避免在查询过程中的任何可能的混淆，它们太相近以致让人十分迷惑。另外，当我表述一张表（基本关系）或者一个视图（派生关系）时，我一般会使用术语“记录集”。

视图是基于关系操作定义的，这是本章的主题。Microsoft Access和SQL Server企业管理器都提供了定义视图的图形化界面，通常也可以通过SQL SELECT语句来定义。

SQL（一般读作“sequel”）表示Structured Query Language（结构化查询语言）。它是表达关系操作的标准语言。Jet数据库引擎和SQL Server都支持SQL语言。当然，它们的形式不同——如果相同就太过简单了。幸运的是，两种不同的实现形式并不影响我们在这一章中讨论的关系代数。当它们语法有所不同时，我们将分别给出这两种形式的例子。

SQL SELECT语句功能十分强大，并且对它更复杂和详尽的研究已超出了本书的范畴。在参考书目中列出了一些常用的参考书籍。出于本书的目的，我们仅限于基本的结构，其语法如下：

```
SELECT <fieldList>
FROM <recordsetList>
    <joinType> JOIN <joinCondition>
WHERE <selectionCriteria>
GROUP BY <groupByFieldList>
HAVING <selectionCriteria>
ORDER BY <orderByFieldList>
```

在SELECT子句中的<fieldList>是一个或多个包含在语句结果记录集中的字段列表。这些字段可以是内在记录集的物理表达，也可以是基于其他字段的计算结果。正如人们所期望的那样，在FROM子句中的<recordsetList>是表和视图的列表，SELECT语句就是建立在这些表和视图基础之上的。只有这两个子句是SELECT语句必须包含的，其他的语句都是可选的。

JOIN子句定义了<recordsetList>中的记录集之间的联系。我们在本章后面将详细讨论连接操作。WHERE子句定义了一个逻辑表达式，<selectionCriteria>，用来限制包含在结果集中的数据。同样，之后我们将详细探讨限制条件。

GROUP BY子句将在特定的字段列上拥有相同值的记录组合成一个单一的记录。当记录被GROUP BY子句组合后，可以用HAVING子句进一步限制返回的行。最后，ORDER BY子句使得记录集按照<orderByFieldList>中列出的字段进行排序。

5.1 Null值和三值逻辑

大多数的关系代数的操作都涉及逻辑运算符的使用，这些运算符通常返回一个布尔结果——True或False。我说“通常”是因为在关系模型中添加了Null值，事情就变得有些复杂了。

Null值给布尔值的集合添加了第三个值，之后必须针对True、False和Null来工作。毫无疑问，这些运算符就被称作三值逻辑。针对标准逻辑运算符的三值真值表如图5-1所示。

正如你看到的，Null与任何值之间的逻辑运算结果都是Null。这一点对逻辑比较运算符也是正确的，如图5-2所示。

AND	True	False	Null
True	True	False	Null
False	False	False	Null
Null	Null	Null	Null

OR	True	False	Null
True	True	True	Null
False	True	False	Null
Null	Null	Null	Null

XOR	True	False	Null
True	False	True	Null
False	True	False	Null
Null	Null	Null	Null

图5-1 三值And, Or和XOr真值表

=	True	False	Null
True	True	False	Null
False	False	True	Null
Null	Null	Null	Null

≠	True	False	Null
True	False	True	Null
False	True	False	Null
Null	Null	Null	Null

图5-2 三值等于和不等值真值表

这里存在两个例外情况。Microsoft Jet在作“True Xor Null”时返回的是True而不是Null（这也许有一定的意义）。可能是出于设计者的考虑，SQL Server给一般的逻辑操作添加了某些“扩展”。如果关掉了ANSI_NULLS，那么Null = Null等于True，而Null = <value>，这里

<value>是除了Null之外的任何值（包括布尔值True和False），其结果为False。坦白地说，我不明白为什么有人想要以这种方式破坏关系代数的规则——在不给运算符的行为增添不确定性前，Null值的处理就已经够复杂了。

SQL提供了两个二元操作符——IS NULL和IS NOT NULL——以便对Null值做一些特殊处理。它们的工作正如人们所期望的一样。图5-3中显示了IS NULL和IS NOT NULL的真值表。同样，<value>是指除了Null之外的任何值。

	Is Null	Is Not Null
<value>	False	True
True	False	True
False	False	True
Null	True	False

图5-3 IS NULL 和IS NOT NULL真值表

5.2 关系运算

在学习关系代数前，让我们先看看四种关系运算：选择、投影、连接和除。选择和投影运算只影响单个记录集，当然一个记录集可以是基于多个其他记录集的视图。连接运算符在关系模型中可能是最基本的，它定义了如何组合两个记录集。最后一个运算符——除，很少用到，但是在决定一个记录集的哪些记录与第二个记录集的所有记录匹配时，就需要用除运算。

所有这些运算符都使用某些形式的SQL SELECT语句来实现。它们可以以任何你想要的形式组合，只需要服从系统关于语句最大长度和复杂度的约束即可。

5.2.1 选择

选择操作符仅返回符合指定选择条件的记录。在SQL SELECT语句中，它是使用WHERE子句的形式实现的，例如以下语句：

```
SELECT *
FROM Employees
WHERE LastName = "Davolio";
```

在Northwind数据库中，该语句返回的是Nancy Davolio的雇员记录，因为她是表中唯一一个姓Davolio的人。（在语句的<fieldList>部分中的*是“所有字段”的缩写。）

在WHERE子句中指定的选择条件可以任意复杂。可以由AND和OR组合的逻辑表达式。该表达式会用来衡量记录集中的每一条记录，并且如果它返回True，则该记录就会被包含在结果中。如果针对记录的表达式返回False或者Null，那么它就不会被包含进来。

5.2.2 投影

选择是对一个记录集的水平切割，而投影则是垂直切割；它仅返回原记录集中字段的一个子集。

SQL在SELECT语句中使用<fieldList>部分实现这个简单的运算，在这个列表中只包含你

所列出的字段。例如，可以使用以下语句来创建一个雇员电话列表：

```
SELECT LastName, FirstName, Extension
FROM Employees
ORDER BY LastName, FirstName;
```

记住，ORDER BY子句仅仅是对数据进行排序，对投影本身没有任何作用和影响。在这个例子中，列表先按姓（LastName字段）的字母排序，然后按名（First Name字段）的字母排序。

5.2.3 连接

连接运算很可能是关系运算中最常见的。当然，它们是模型的基础——如果将数据分解成多个关系是不可行的话，就不可能有必要将它们组合起来。这正是一个连接运算符所做的事情：它在一个或多个公共字段的比较的基础上组合记录集。

连接运算是使用SELECT语句的JOIN子句实现的。它根据字段比较的类型以及比较结果的处理方式进行分类。我们依次来讨论每一种类别。

5.2.3.1 等值连接

当连接比较是基于相等时，该连接就叫做**等值连接**（equi-join）。在等值连接运算中，只返回那些在指定的字段有匹配值的记录。

以图5-4中的关系为例来说。这是典型的规范化以后产生的关联表。OrderID是Orders表的主码，并且是Order Details表的外码。

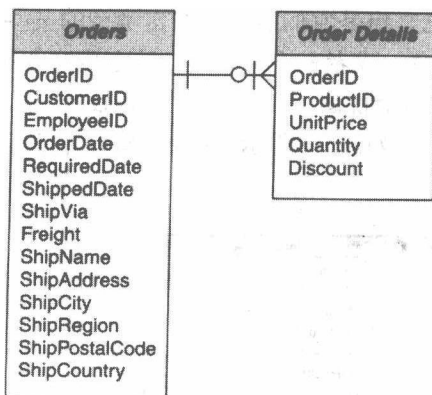


图5-4 这两张表可以通过JOIN运算符组合起来

为了组合（结果非规范化了）这两张表，可以使用下面的SELECT语句：

```
SELECT Orders.OrderID, Orders.CustomerID, [Order Details].ProductID
FROM Orders
INNER JOIN [Order Details] ON Orders.OrderID = [Order Details].OrderID
WHERE Orders.OrderID=10248;
```

该语句会生成如图5-5所示的结果集。

注意：如果你在Access 2000的Northwind 数据库中运行这个查询，结果集会显示客户的名，而不是CustomerID。这是因为当你在表定义中声明了查询控制后，Access允许字段显

示某些文字而不是它们实际存储的内容。当Access用于交互时，这是十分方便的，但是对于作者提供的例子会十分不恰当。

5.2.3.2 自然连接

一种特殊的等值连接是自然连接。要符合自然连接，连接运算必须满足以下条件：

- 比较运算符必须是等号。
- 所有的公共字段都必须参与到连接中。
- 必须在结果记录集中包含一个公共字段集。

自然连接没有什么奇妙之处。它们没有以特殊的方式行事，数据库引擎也没有为它们提供特殊的支持。它们是相当常见的连接形式，以至于给它们起这样一个自然连接的名字。

如果你创建一个满足额外条件的自然连接，Jet数据库引擎会做一些特别的事情。如果两个表之间建立了一对多的联系，并且包含在视图中的公共字段全部来自联系的“多”端，那么Jet数据库引擎会实施称作RowFix-Up或者AutoLookup的操作。当用户在一个Access窗体的控件中输入了一个值，而且该控件绑定到了连接条件中使用的字段，那么Jet数据库引擎将自动为那些与“多”端的字段绑定的控件提供数据。这是一个绝好的技术，使得程序员的工作简单了很多。

5.2.3.3 θ 连接

建立在任何除等号之外的比较符（ $<$ ， $>$ ， $>=$ ， $<=$ ）基础上的连接称作 θ 连接。（从技术角度来说，所有的连接都是 θ 连接，但是习惯上，如果比较符是等号，那么该连接通常认为是等值连接或者就是一个连接。）

θ 连接在实际应用中很少见，但是它们能解决特定类型的某些问题。这些问题大多涉及查询值大于平均值或者总值的记录，或者查询落在某个特定范围内的记录。

举个例子，比如你创建了两个视图，第一个包含了每个产品类别的销售量平均值，第二个包含了每种产品的销售总和，如图5-6所示。（我们将在本章后面探讨如何创建这些视图。现在就假定它们已经存在。）

Category ID	Category Name	AverageSold
1	Beverages	24
2	Condiments	25
3	Confections	24
4	Dairy Products	25
5	Grains/Cereals	23
6	Meat/Poultry	24
7	Produce	22
8	Seafood	23

Category ID	Product ID	Product Name	TotalSold
1	1	Chai	828
1	2	Chang	1057
1	24	Guaraná Fantástica	1125
1	34	Sasquatch Ale	506
1	35	Steeleye Stout	883
1	38	Côte de Blaye	623
1	39	Chartreuse verte	793
1	43	Ipoh Coffee	580
1	67	Laughing Lumberjack Lager	184
1	70	Outback Lager	817
1	75	Rhönbräu Klosterbier	1155
1	76	Lakkalikööri	981
2	3	Aniseed Syrup	328

图5-6 这些视图能够使用一个 θ 连接连接起来

Order ID	Customer	Product
10248	WILMK	11
10248	WILMK	42
10248	WILMK	72

图5-5 这个记录集是Orders和Order Details表连接后的结果

下面的SELECT语句是基于比较符“>”的，它将产生在每种类别中的最佳销售产品的列表：

```
SELECT DISTINCTROW ProductCategoryAverages.CategoryName,
    ProductTotals.ProductName
FROM ProductCategoryAverages
INNER JOIN ProductTotals
    ON ProductCategoryAverages.CategoryID = ProductTotals.CategoryID
    AND ProductTotals.TotalSold > [ProductCategoryAverages].[AverageSold];
```

其结果如图5-7所示。

Category Name	Product Name
Beverages	Chai
Beverages	Chang
Beverages	Guaraná Fantástica
Beverages	Sasquatch Ale
Beverages	Steeleye Stout
Beverages	Côte de Blaye
Beverages	Chartreuse verte
Beverages	Ippoh Coffee
Beverages	Laughing Lumberjack Lager
Beverages	Outback Lager
Beverages	Rhônebräu Klosterbier
Beverages	Lakkalikööri
Condiments	Aniseed Syrup
Condiments	Chef Anton's Cajun Seasoning

图5-7 本记录集是一个 θ 连接的结果

在这个例子中，这个视图也可以使用WHERE子句的限制条件来定义。事实上，如果SQL视图如以下形式，则Access将会重写该查询：

```
SELECT DISTINCTROW ProductCategoryAverages.CategoryName,
    ProductTotals.ProductName
FROM ProductCategoryAverages
INNER JOIN ProductTotals
    ON ProductCategoryAverages.CategoryID = ProductTotals.CategoryID
WHERE ProductTotals.TotalSold>ProductCategoryAverages.AverageSold;
```

从技术的角度来说，所有的连接，包括等值连接和自然连接，都能用一个限制来表述。在 θ 连接的例子中，这种使用一个限制来表达的形式总是比较受青睐的，因为数据库引擎能够更好地优化它的执行。

5.2.3.4 外连接

自此我们学习的所有连接都是内连接，这些连接仅仅返回那些使得连接条件为True的记录。注意这与仅返回那些与指定的字段相匹配的记录不完全相同，虽然通常一个内连接是这样描述的。“匹配”意味着相等，而我们就所知道的，并不是所有的连接都是基于相等的。

关系代数还支持另外一种连接——外连接。一个外连接返回所有内连接返回的值，再加上一边或者两边记录集的所有记录。失去的（“不匹配”）的值置为Null。

外连接可以分为左外连接、右外连接或者全外连接，这要根据将要包含哪些额外的记录来定。现在，当我还在高校时，左外连接返回所有的处于一对多联系中的“一”端的全部记录集，而右外连接则返回所有“多”端的记录集。但是，对于Jet数据库引擎和SQL Server，左、右外连接的区别在于列在SQL SELECT语句中的记录集的顺序。因此，下面两个语句均返回的是X的所有记录和那些使得<condition>等于True的Y中的记录。

```
SELECT * FROM X LEFT OUTER JOIN Y ON <condition>
SELECT * FROM Y RIGHT OUTER JOIN X ON <condition>
```

一个全外连接返回两个记录集的所有记录，包括那些使得条件等于True的记录。SQL Server使用FULL OUTER JOIN条件支持全外连接：

```
SELECT * FROM X FULL OUTER JOIN Y ON <condition>
```

Jet数据库引擎不直接支持全外连接，但是通过实施一个左外连接和一个右外连接的合并就能实现全外连接了。我们将在下一节讨论合并。

5.2.4 除

最后一个关系运算是除。**关系除运算符**（如此称呼是为了与数学除区分开来）返回一个记录集中值与第二个记录集中所有对应值相匹配的记录。例如，给定一个从每个供应商那里购买的产品类别的记录集，一个关系除可以生成一个提供所有种类产品的供应商列表。

这并不是一个罕见的情况，但是其解决方法不够直观，因为SQL SELECT语句不直接支持关系除。但是，有很多方式能达到与关系除相同的结果。最简单的方法就是重新描述该查询。

不用“列举提供所有产品类别的供应商”，这较难处理，而表述为“列举那些提供的产品类别数与所有产品类别数相同的供应商”。这是一个扩展运算的例子，我们将在本章后面讨论。它并不是总能起作用，在该方法无效的情况下，可以使用关联查询来实现除。但是，关联查询超出了本书的范畴。请查看本书列出的参考书目中的有关参考书。

5.3 集合运算符

接下来的四个关系代数运算符是基于传统集合理论的。但是，由于我们是处理关系，不是无差别的集合，因此这些运算也进行了一些小小的修改以适应关系的操作，毕竟关系与集合略有差别。

5.3.1 并

从概念上来讲，一个**关系并**是两个记录集的拼接。它或多或少类似于关系中添加的形式。记录集A和记录集B的并的结果实际上与将A中的所有记录添加到B中一样。

举个例子，为了大量的邮寄，你需要数据库中所有名字和地址的列表。Northwind数据库中Customers和Employees记录集都有地址，因此通过一个并运算就能简单地将它们合并在一起。在该例子中，我们使用UNION语句，如下所示：

```
SELECT CompanyName AS FullName, Address, City, PostalCode
FROM Customers
UNION SELECT [FirstName] & " " & [LastName] AS FullName,
            Address, City, PostalCode
FROM Employees
ORDER BY FullName;
```

注意，CompanyName字段被重命名为“FullName”，并且Employees表中的FirstName和LastName字段被连接到了一起。其结果字段也称作“FullName”。并查询不要求每一个SELECT语句中的<fieldList>中的字段都有相同的名字，但是它们必须有相同的数目，而且它们的类型必须相同（或者兼容）。在Access中该语句的结果如图5-8所示。

Name	Address	City	PostalCode
Alfreds Futterkiste	Obere Str. 57	Berlin	12209
Ana Trujillo Emparedados y helados	Avda. de la Constitución 2222	México D.F.	05021
Andrew Fuller	908 W. Capital Way	Tacoma	98401
Anne Dodsworth	7 Houndstooth Rd.	London	WG2 7LT
Antonio Moreno Taquería	Mataderos 2312	México D.F.	05023
Around the Horn	120 Hanover Sq.	London	WA1 1DP
Berglunds snabbköp	Berguvsvägen 8	Luleå	S-958 22
Blauer See Delikatessen	Forsterstr. 57	Mannheim	68306
Blondel père et fils	24, place Kléber	Strasbourg	67000
Bólido Comidas preparadas	C/ Araquil, 67	Madrid	28023
Bon app'	12, rue des Bouchers	Marseille	13008
Bottom-Dollar Markets	23 Tsawassen Blvd.	Tsawassen	T2F 8M4

图5-8 UNION语句组合了来自两个表的记录

5.3.2 交

关系交运算返回的是在两个记录集都有的共同的记录。实际上，它是一个“寻找重复项”的操作，并且这也是它经常被使用的方式。

可以使用外连接来实现交运算。作为一个例子，假定你从数个过去遗留的系统中得到了客户列表，如图5-9所示。

CustomerID	CompanyName
ALFKI	Alfreds Futterkiste
ANATR	Ana Trujillo Emparedados y helados
ANTON	Antonio Moreno Taquería
AROUT	Around the Horn
BERGS	Berglunds snabbköp
BLAUS	Blauer See Delikatessen
BLONP	Blondel père et fils
BOLID	Bólido Comidas preparadas

CustomerID	CompanyName
ALFKI	Alfreds Futterkiste
ANATR	Ana Trujillo Emparedados y helados
ANTON	Antonio Moreno Taquería
AROUT	Around the Horn
FAMIA	Familia Arquibaldo
FISSA	FISSA Fabrica Inter. Salchichas S.A.
FOLIG	Folies gourmandes
FOLKO	Folk och fa HB
FRANK	Frankenversand

图5-9 遗留表中通常存在重复数据

下面的SELECT语句将会返回这些重复记录：

```
SELECT DuplicateCustomers1.*
FROM DuplicateCustomers1
LEFT JOIN DuplicateCustomers2
    ON DuplicateCustomers1.CustomerID = DuplicateCustomers2.CustomerID
WHERE DuplicateCustomers2.CustomerID IS NOT NULL;
```

这个语句的结果如图5-10所示。

5.3.3 差

两个记录集的交被用来“寻找重复项”，而差运算符则是用来“寻找孤立项”。两个记录

集的关系差返回的是属于一个记录集但不属于另一个记录集的记录。

作为例子，同样给定如图5-9所示的两个记录集。下面的SELECT语句将返回不匹配的记录：

```
SELECT DuplicateCustomers1.*
FROM DuplicateCustomers1
LEFT JOIN DuplicateCustomers2
ON DuplicateCustomers1.CustomerID = DuplicateCustomers2.CustomerID
WHERE DuplicateCustomers2.CustomerID IS NULL;
```

CustomerID	CompanyName
ALFKI	Alfreds Futterkiste
ANATR	Ana Trujillo Emparedados y helados
ANTON	Antonio Moreno Taquería
AROUT	Around the Horn

图5-10 一个外连接加上IS NOT NULL运算符就能实现交运算

这个语句中的外连接运算将返回两个列表中的所有记录。正如你所会想到的，一个外连接会给那些在另一个表中没有匹配项的字段提供Null值。WHERE子句使用IS NULL运算符限制仅返回那些不匹配的记录。

如果这样不够清楚，那么试一试分两步来实现该操作：第一步以视图的形式创建一个外连接，然后使用WHERE子句来限制该视图。这个过程如图5-11所示。

Step 1: Create the outer join

```
SELECT DuplicateCustomers1.*
FROM DuplicateCustomers1
LEFT JOIN DuplicateCustomers2
ON DuplicateCustomers1.CustomerID = DuplicateCustomers2.CustomerID
```

05_09a.Custom	CompanyName	05_09b.Custom
ALFKI	Alfreds Futterkiste	ALFKI
ANATR	Ana Trujillo Emparedados y helados	ANATR
ANTON	Antonio Moreno Taquería	ANTON
AROUT	Around the Horn	AROUT
BERGS	Berglunds snabbköp	
BLAUS	Blauer See Delikatessen	
BLONP	Blondel père et fils	
BOLID	Bólido Comidas preparadas	

Step 2: Select for CustomerID = Null

```
SELECT DuplicateCustomers1.*
FROM DuplicateCustomers1
LEFT JOIN DuplicateCustomers2
ON DuplicateCustomers1.CustomerID = DuplicateCustomers2.CustomerID
WHERE DuplicateCustomers2.CustomerID is Null
```

05_09a.Custom	CompanyName	05_09b.Custom
BERGS	Berglunds snabbköp	
BLAUS	Blauer See Delikatessen	
BLONP	Blondel père et fils	
BOLID	Bólido Comidas preparadas	

图5-11 分两步实现差运算

5.3.4 笛卡儿积

最后一个集合运算是笛卡儿积。与传统的集合理论类似，两个记录集的笛卡儿积是将一个集合中的每个记录与另一个集合中的每个记录相组合。

两个记录集的笛卡儿积（或者就是“积”）可以通过一条仅指定两个（或多个）记录集、而没有JOIN子句的SELECT语句实现。下面的语句将返回每个客户与每个客户服务代表的组合：

```
SELECT CustomerName, CSRName FROM Customer, CSRs;
```

有时，笛卡儿积为了分析的目的或者作为进一步操作的中间结果时十分有用，尽管它们大多数时候都只是偶然产生的。比如，在Access查询设计器中忘记画连接线，那么你将得到一个笛卡儿积。它相当容易获得，所以当你第一次碰到时请不要觉得奇怪。

5.4 特殊的关系运算符

当关系模型首次被明确阐明的时候，关系代数的各种形式的扩展就被提出了。我们将关注三个普遍接受的形式：总结（summarize）、扩展（extend）以及重命名（rename）。我们还将关注由微软提供的三种扩展形式：变换（transform）、上卷（rollup）和立方体（cube）。

5.4.1 总结

总结运算符所做的正是人们所期望的：它产生包含根据特定字段分组后的总结数据的记录。如果你想在更高的抽象级别检查数据，而不是数据存储的级别，那么总结运算在这些情况下十分有用。

总结运算通过使用SELECT语句的GROUP BY子句实现。它对指定字段或多个字段中的每个值，返回一条记录。如果列出了多个字段，那么分组将会嵌套。例如，考虑下面的语句：

```
SELECT Categories.CategoryName, Products.ProductName,
       SUM([Order Details].Quantity) AS SumOfQuantity
FROM Categories
INNER JOIN Products
    ON Categories.CategoryID = Products.CategoryID)
INNER JOIN [Order Details]
    ON Products.ProductID = [Order Details].ProductID
GROUP BY Categories.CategoryName, Products.ProductName;
```

这个语句将会为Northwind数据库中的每个产品返回一条记录，它根据类别分组，并且包含三个字段：CategoryName、ProductName以及SumOfQuantity——每种产品的销售总数，如图5-12所示。

Category Name	Product Name	SumOfQuantity
Beverages	Chai	828
Beverages	Chang	1057
Beverages	Chartreuse verte	793
Beverages	Côte de Blaye	623
Beverages	Guaraná Fantástica	1125
Beverages	Ispoh Coffee	580
Beverages	Lakkalikööri	981
Beverages	Laughing Lumberjack Lager	184
Beverages	Outback Lager	817
Beverages	Rhônebräu Klosterbier	1155
Beverages	Sasquatch Ale	506
Beverages	Steeleye Stout	883

图5-12 GROUP BY子句返回总和数据

在SELECT语句的<fieldList>中列出的字段必须或者是<groupFieldList>中的一部分，或者是一个SQL统计函数的一个参数。SQL统计函数为每一条记录计算总和值。最常见的统计函数有AVERAGE、COUNT、SUM、MAXIMUM和MINIMUM。

统计函数是另一个因为Null而头疼的地方。Null值会包含在总结运算中——它们也形成了一组。但是，它们会被统计函数忽略掉。如果使用<groupFieldList>中的一个字段作为统计函数的一个参数，那么这通常是你遇到的唯一问题。

5.4.2 扩展

扩展运算符允许你定义一个虚拟字段，它是由存储于数据库中的常量或者值计算得到的。可以通过在SELECT语句的<fieldList>部分定义它们来创建虚拟字段，如下：

```
SELECT [UnitPrice]*[Qty] AS ExtendedPrice
FROM [Order Details];
```

定义在虚拟字段中的计算可以任意复杂。这个过程简单而且快速，因此很少有理由在表中存储一个计算字段。

5.4.3 重命名

最后一个常见的运算符是重命名。重命名运算可以运用在<recordsetList>中的一个记录集上，也可以用在<fieldList>的每个字段上。在Jet数据库引擎中，一个记录集的重命名使用如下语法：

```
SELECT <fieldName> AS <fieldAlias>
FROM <tableName> AS <tableAlias>
```

在SQL Server中，“AS”关键字不是必需的，如下所示：

```
SELECT <fieldName> <fieldAlias> FROM <recordsetName> <recordsetAlias>
```

当需要定义一个有自连接的视图时，重命名会特别有用，如下所示的代码：

```
SELECT Manager.Name, Employee.Name
FROM Employees AS Employee
INNER JOIN Employees AS Manager
    ON Employee.EmployeeID = Manager.EmployeeID;
```

这个语法可以使每个表在逻辑上是不同的。

5.4.4 变换

TRANSFORM（变换）语句是我们将要学习的微软对于关系代数的第一个扩展。TRANSFORM使用总结（GROUP BY）运算的结果，并将它们旋转90度。更多的时候它作为一个交叉表查询，这无疑是个非常有用的运算，但只有Jet数据库引擎支持这种运算，在SQL Server中尚未实现。

YUKON注意：SQL Server 所支持的Transact-SQL语言已经在YUKON中得到了扩展，并支持变换运算。

TRANSFORM语句有如下的基本语法:

```
TRANSFORM <aggregateFunction>
SELECT <fieldList>
FROM <recordsetList>
GROUP BY <groupByList>
PIVOT <columnHeading> [IN (<valueList>)]
```

TRANSFORM 的<aggregateFunction>子句定义了填充记录集的总结数据。SELECT语句必须包括一个GROUP BY子句而且不能有HAVING子句。在任何GROUP BY子句中,<groupByList>可以包含多个字段。(在一个变换语句中,<fieldList>和<groupByList>表达式通常都是确定的。)

PIVOT子句标识其值将要作为列标题的字段。默认情况下,Jet数据库引擎将在记录集中按字母顺序包含所有列。不过,可选的IN语句允许你指定列名,它将按照它们在<valueList>中的顺序列出。

下面的TRANSFORM语句实质上提供了与之前给定的总结运算的例子相同的信息,其结果如图5-12所示。

```
TRANSFORM Count(Products.ProductID) AS CountOfProductID
SELECT Suppliers.CompanyName
FROM Suppliers
INNER JOIN (Categories INNER JOIN Products
ON Categories.CategoryID = Products.CategoryID)
ON Suppliers.SupplierID = Products.SupplierID
GROUP BY Suppliers.CompanyName
PIVOT Categories.CategoryName;
```

这个变换运算的结果如图5-13所示。

Company Name	Beverages	Condiments	Confections	Dairy	Grains/Cereals	Meat/Poultry	Produce	Seafood
Aux joyeux ecclésiasti	2							
Bigfoot Breweries	3							
Cooperativa de Quesos				2				
Escargots Nouveaux								1
Exotic Liquids	2	1						
Forêts d'érables		1	1					
Formaggi Fortini s.r.l.				3				
Gai pâturage				2				
G'day, Mate					1		1	1
Grandma Kelly's Home		2					1	
Heli Süßwaren GmbH			3					
Karkki Oy	1		2					
Leka Trading	1	1			1			
Lyngbysild								2

图5-13 变换语句将结果旋转了90度

5.4.5 上卷

总结运算符是使用GROUP BY子句实现的,它生成了包含总结数据的记录。ROLLUP子句通过提供总和和数据为这个操作提供了一种逻辑扩展。但只有SQL Server支持ROLLUP子句。

ROLLUP子句仅仅在SQL Server中可行,它是作为GROUP BY子句的一种扩展实现的:

```
SELECT Categories.CategoryName, Products.ProductName,  
       SUM([Order Details].Quantity) AS SumOfQuantity  
FROM (Categories INNER JOIN Products  
      ON Categories.CategoryID = Products.CategoryID)  
INNER JOIN [Order Details]  
      ON Products.ProductID = [Order Details].ProductID  
GROUP BY Categories.CategoryName, Products.ProductName WITH ROLLUP;
```

5.4.6 立方体

CUBE运算符只有在SQL Server中可行，它也是GROUP BY子句的一种扩展。本质上，CUBE子句是通过其他所有的列总结了<groupByList>中的每个列。在概念上它类似于ROLLUP运算符，但是ROLLUP值为在<groupByList>中指定的每一列生成总和，而CUBE为其他的组创建总结数据。

例如，如果在<groupByList>中有三个字段——A，B和C，那么CUBE运算符将会返回如下七个统计值：

1. 所有C的总数。
2. 根据A分组后，C的总数。
3. 在A中根据C分组后，C的总数。
4. 在A中根据B 分组后，C的总数。
5. 根据B分组后，C的总数。
6. 在B中根据A分组后，C的总数。
7. 在B中根据C分组后，C的总数。

5.5 小结

本章我们介绍了使用不同的关系运算符来操作基本关系的方法，并且举例介绍了如何在SQL语言中实现这些关系运算。此外，还讨论了Null和三值逻辑问题。

在标准的关系运算符中，选择和投影运算符是从单个记录集中选择子集。而连接、并、交、差和笛卡儿积运算符都是控制两个记录集组合的方式。所有这些运算符，除了差运算符之外，都能通过SQL SELECT语句实现。差运算有时候可以使用SELECT语句实现，而有时候需要其他的技术，这一点超出了本书的范畴。

我们还讨论了一些特殊的运算符。总结和扩展运算符对数据实施了计算。重命名运算符控制显示在视图中的列标题。变换、上卷和立方体是由微软实现的对SQL语言的特殊扩展，它们每一种也都提供了总结和查看数据的特殊方式。

在回顾了关系代数之后，我们就结束了第一部分。当然，关系数据库的理论是复杂的，因此，作为一本介绍性的书，我们并没有将所有的问题都包含进来。但是你现在已经看到了该理论的全部主要内容。在下一节，我们将转向关系数据库的另一类模型：多维模型。

第二部分 维度数据库理论

第6章 维度的基本概念

在第一部分，我们讨论了传统关系理论和实体联系模型的基本原则。大部分的数据库应用程序都使用这些原则构建，并且有很好的效果。如果始终如一地应用它，那么可以确保那些数据频繁变动的应用程序的数据完整性、可恢复性和可度量性。

但是不是所有的数据库应用都涉及频繁的数据变动，存在另一种完全不同的数据库应用，它有完全不同的目的和要求。这些应用必须存储大量（有时候是海量的）的历史数据，但是这些数据一旦存储了，就很少改变。这些应用很少支持大量用户，但是这些用户都必须能利用这些数据有效地生成报表。“有效地”在这种情况下有两重含义：特别的请求必须迅速返回结果，并且数据库模式的结构必须易于最终用户的理解。

使用我们在第一部分介绍的规范化所生成的数据库模式很难满足这些要求。即使数据库足够小，而且响应迅速，也很少能使用关系模型处理好历史数据。另外，完全规范化的模式很难满足“易于最终用户理解”的要求。

在这一节，我们将学习维度数据库的规则，它们将能满足这些需求。本章中，我们主要关注维度数据库的基本结构。

6.1 维度数据库模型

基于实体联系模型的关系型数据库通常支持一个企业操作方面的事务处理，例如订单处理或者薪水管理。它们回答的问题是：“Mary Smith在6月5日订购了多少螺丝刀？”或者“2003年1月，我们需要付给Jimmy Jones多少薪水？”。这些数据库中的数据“粒度”是单个事务——一行单一的订购项（螺丝刀的数目），或者一张薪水存根（Jimmy Jones 1月份的薪水），因此这样的系统通常被称作联机事务处理（On-Line Transaction Processing, OLTP）系统。

这种操作十分有用，但是它不够准确，因为基于实体联系模型的模式也是存储静态数据的最好的结构——例如，一个图书馆的当前藏书情况（“卡片目录”），或者一个邮寄列表。

另一方面，基于多维模型的数据库模式通常用来分析一个企业的状态。它们回答的问题是：“哪些产品组利润最大？”或者“各季节的销售变化是什么样的？”出于这样的原因，它们常被称作联机分析处理（On-Line Analytical Processing, OLAP）系统。

同样，OLAP也是一个十分有用但是不够准确的术语。OLAP系统除了分析外，很少用于其他事务，而OLTP系统也常常用于分析。因此，这两种类型的系统并不是相互排斥的。进一步来说，如同保持数据库的逻辑设计和物理设计的不同性一样，我们需要一些功能性的术语来保持这两种系统中结构上的差异性。

出于这些原因，我们打算使用术语“规范化”和“多维的”来区分使用这两套规则开发

的模式。但是要注意这是我个人的术语。在很多文献中，这两种类型的数据库模式很少一同讨论，因此就没有必要做区分。我所称的规范化数据库简单的就是指“数据库”，而多维的数据库是指“维度数据库”或者“OLAP数据库”。

但是，出于本书的目的，我们有必要区分这两种类型的数据库。所以，我将使用术语**规范化数据库**来表示我们在第一部分学习的传统的结构，它定义为以下几个方面：实体、属性和联系，如图6-1所示。

注意在图6-1中的两个实体并不是一个完整的模式；Customer、Employee和Product实体都没有列出。这比完整的数据库要易于理解一些。（我曾经和一位分析员共事，她创建最初模式时仅使用小小的黄色粘贴纸和线条。她设计的其中一个模式占据了12英尺长的走廊的最佳位置。请将这一点教授给市场部主管！）

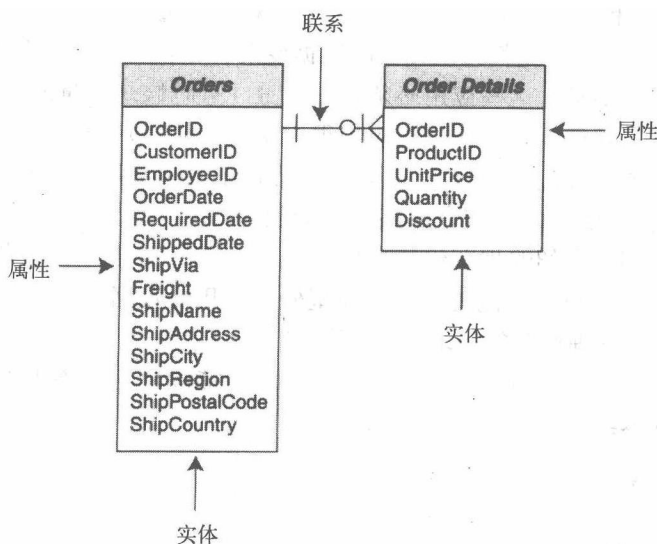


图6-1 来自规范化数据库模式的两张表

图6-2显示了与图6-1等价的**维度数据库**，它是由事实表、事实表和维来定义的。正如你所看到的，这两种模型之间有一些重叠。毕竟，我们仍旧是在处理关系数据库，因此，该模式仍然是由逻辑关系（虽然它们在高维设计中通常被称作“表”）组成的，但是这些关系被分为了两种不同的类型：一个事实表和多个维度表。

注意，与规范化形式不同，图6-2在模式中包含所有的关系。为了可读性，维度表没有显示出它的所有属性，但是所有的维度都显示出来了。对于最终用户，这种结构比等价的规范化模式好理解得多。

单个的**事实表**是对用来生成报表的数据的建模。它包含两种不同类型的属性：**关键属性**——所有这些都是连接事实表和维度表的外码；**事实**——该属性包含用来度量的事实数据。

注意事实表在该图形中被置于中间，维度表安排在其周围。这是这些模式的一种方便的排列方式。因此，有人认为这种排列方式像一颗星星，从而称之为**星型模式**。

我得承认在这里我没有看出它们有任何相似之处，可能我在做数据库设计时遗忘了我的诗意（也可能我更像一个分析者，而不是设计人员）。不管怎样，无论这个图形是否排列为这种样式，它都被称作星型模式。

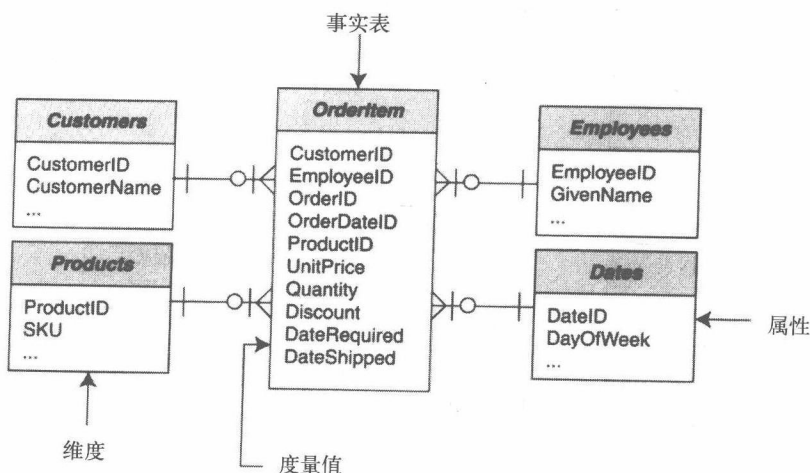


图6-2 一个多维模式

维度数据库真是关系的吗？

并不总是的。这里有三种存储维度数据库的物理方式：ROLAP、MOLAP和HOLAP。ROLAP代表关系OLAP。使用ROLAP模型的维度数据库将其数据存储在一个关系数据库引擎中，比如SQL Server，因此它比较符合“关系数据库”的说法。

MOLAP代表多维OLAP。这些结构不是关系的。例如，SQL Server Analysis Services为MOLAP数据库提供了一种不同的物理存储结构。最后，HOLAP表示混合OLAP。这种数据库使用关系和多维存储的组合方式。

事实上，也并不是所有的规范化数据库在物理存储时都使用关系引擎。大部分使用实体联系模式开发的数据库模式都是使用一个关系引擎，但并不是所有都如此。现在这些数据库越来越多的使用XML作为数据存储机制。（XML是层次的，不是关系的。）

实体联系模型本质上依旧是关系的。尽管维度数据库的物理实现可能不是关系的，但多维模型其实本质上也是关系的。

记得我说重要的是保持逻辑设计和物理设计的区别吗？那时你认为我在开玩笑……

定义一个星型模式的技术是：它由一个事实表和多个维度表组成，所有维度表都直接与事实表相连接，并且作为一个通用规则，它们不能与其他维度连接。但是，有时也存在一些情况需要将维度划分为多个关系，如图6-3所示。

这种结构被称为是雪花模式。雪花模式是规范化维度的结果。作为一般规则，规范化不能简单地应用于维度数据库，但是也存在一些例外情况，我们将在第8章讨论。

另一种将数据存储于维度数据库中的方法就是把它作为一个立方体，如图6-4所示。的确，很多人都认为维度数据库就是一个“立方体”。事实上，这是微软在其文档和产品中使用的术语，并且我们将用它来表示物理数据存储。

从技术角度来说，它应当是一个“N-立方体”，因为它可以有超过三个的维度，但是“N”通常被省去。（你很可能想起这与n-元组相同，而n-元组通常也就简单地认为是元组。）

使用术语“立方体”并不意味着一个不同的数据库模式——模式仍旧是由一些规则定义的，

我们将在后续章节学习到——它仅仅反映了一种不同的考虑数据的方式。

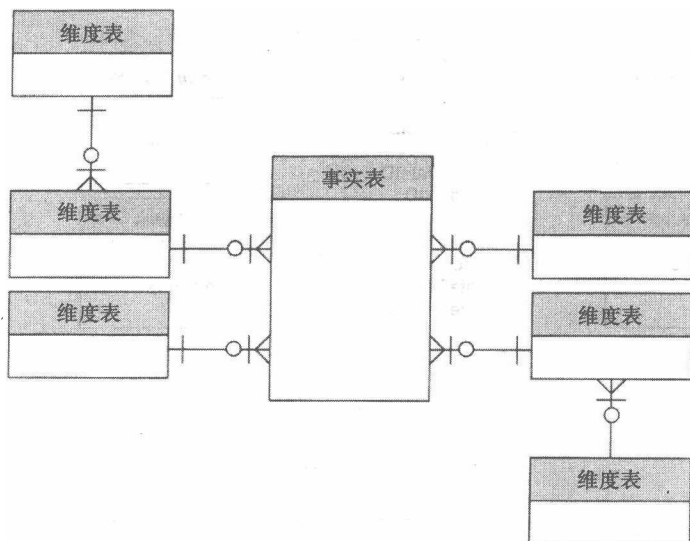


图6-3 一个雪花模式

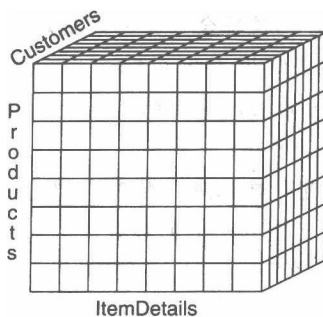


图6-4 一个多维立方体

6.2 术语

在多维设计技术里有一套应用程序架构和分析过程，它们大多数都超出了本书的范畴，但我们还是要简单的介绍一下这些概念，以使多维分析在本书中上下文关联。

不幸的是，这个领域是全新的，并且术语还没有被完全定下来——不过我们在本节讨论的大部分术语都已经被用来描述整个领域。和往常一样，我们在本书的内容中将挑选一些比较有意义的（对我而言）和常用的术语，但是要注意其他来源的说法并不一定表示相同的含义——一定要检查一下。

例如，微软通常使用术语“商务智能”（business intelligence）来描述这个领域，有时还使用“数据仓库”（data warehousing）来描述——事实上，他们用它来描述多项事物。其他作者使用“数据仓库”作为一般术语而根本不使用“商务智能”。在本书的内容里，我将使用商务智能来表示整个领域。

我们将使用**数据仓库**来代表其物理数据结构，使用OLAP来表示其应用。我们称每个数据立方体为**数据集市**。使用这个定义，一个数据仓库可以被认为是一个企业使用的每个数据集市的集合。注意其他作者可能会不同地使用这些术语。数据仓库有时用来描述数据立方体的源数据。

在文献里，术语**数据挖掘**通常指的是对事务性数据的特定统计技术的应用，这些应用是为了抽取预测性的规则。数据挖掘是一个精确的数学领域。（SQL Server 2000提供了一些具体的技术。）它并没有包揽所有使用数据仓库的分析任务。我们将**商务分析**作为一个一般术语使用。如果你期望用一个更特殊的术语，（那么我很抱歉。但是我从未看过当已经存在一个恰当的术语时，却使用新异术语会有什么好处。）

YUKON注意：SQL Server YUKON在产品中增添了一些新的重要的数据挖掘功能，还有一个全新的分析服务的界面。你可以期待本章中的信息仍旧可用，但是它将在新版本中有所扩展。

为了获得完整的企业概貌，通常需要从多个OLTP系统中抽取数据。例如，一个企业使用两个OLTP系统，一个用于操作，另一个用于订单处理。最实际的解决方案是分别构建两个不同的数据立方体，每个系统一个。

6.3 商务智能的浓缩历史

所有这些技术和过程，包括多维设计，都是来自于需求。当然，这是十分显然的。但是令人惊奇的是，它在我们的行业里却很少见——更多的时候则认为需求滞后了技术。无论如何，一旦大多数操作型的系统计算机化之后，人们意识到存储在OLTP系统中的数据可能会在管理层的决策制定过程中十分有用，但是这些数据是不利于分析的。

打印报表是基于OLTP数据提供信息的传统方法。不幸的是，它很难提前预测分析报告中的影响因素。专门的报表是有问题的，大多数报表工具存在的问题是没有面向最终用户。我们不会阻止你建议一个大公司的市场经理学习Crystal Reports或者Microsoft Access报表生成器，就像报表书写人员一样。这两个工具都是十分出色的。

当然，Microsoft Excel是最常见的。我敢打赌至少有90%的商业分析是使用Microsoft Excel完成的，只是因为它是市场上唯一的产品。它能够完成我们要求的工作，但是不如人们想象的那样有效。（我得承认我在这里有一定的偏见。Excel是一个数学工具，而不是一个数据工具，而且我十分愤恨我一生大量的时间都在给人们解释这一点。）

首先，我们仍然有规范化模式的问题。如果将商业分析员和经理们都限制在预定义的查询上，那么你很可能使得分析过程不可接受。但如果你让这些用户拥有对这些原始数据的完全访问权，那么你会冒着破坏系统的风险。

记住，在查询中的OLTP系统是用作事务处理的。这就意味着如果一名分析人员对三张表作笛卡儿积运算，那么将会给企业带来完全暂停的后果。即使是在最好的环境里，大部分分析人员和经理想要查询的问题都要求大量（速度慢）的连接，这给系统造成了不可接受的负担。

很显然，不可能因为系统太忙而不响应关于西南地区的订单请求，就阻止在西南地区的用户输入订单。

数据仓库就是设计来处理这些问题的。在一个数据仓库中，数据是从多个OLTP系统中抽取出来的，并且经过“清理”使得它们以简单易懂的形式展现给分析者和经理们。多维设计表达了这种设想，被认为是表达那种形式的数据的最佳方法。

直到SQL Server 2000面世前，数据仓库都是模糊、昂贵的，并且常常只限于大型的企业。多维分析不被认为是关系数据库设计的主流部分，尽管大量的数据仓库都是在关系数据库上实现的。

但是和通常情况一样，如今商务智能已经可以应用在桌面上了，它也被集成到主流的应用设计之中。我们强烈建议你在合适的地方考虑使用它——即使是小型企业也可以从更好的理解它们的操作中受益。

并且我可以承诺我并非受雇于微软的市场部。很简单，我已经太老了，足够理解这是怎样的一个革命……就像用文字处理软件来替代打字机一样。（而且，我也足够老的能记住它！）

6.4 小结

在这一章中，我们以较高的层次关注了维度数据库。我们看到，在前几章节学的传统的规范化数据库是设计用来处理频繁的数据添加和更新的，而维度数据库主要是针对最终用户的特殊查询。数据一旦上载到维度数据库，就很少更新了。星型和雪花型模式与用来定义规范化数据库的实体联系图等价。星型模式由事实表和维度表来定义关系。事实表由两种类型的属性组成：关键属性，是维度表的外码；事实，是存储的事实数据。

由星型模式定义的关系也可以认为是一个立方体（技术上称为“n-立方体”）。数据立方体和星型模式基本上是可以互换的，它们是用来描述同一概念的不同表达方式。

多维设计是众所周知的商务智能领域（也被一些作者称为“数据仓库”）的分析组件。用这些技术构建的应用程序有时候被称为OLAP（“联机分析处理”）应用程序，以便与利用实体联系技术构建的OLTP应用程序区分开来。

OLAP应用程序常常使用数据挖掘技术来决定所分析数据的趋势。但是，数据挖掘仅仅是使用这些应用程序实施的商业分析的一个组件。

在第7章，我们将开始更细致的学习维度数据库中对事实表的定义。

第7章 事实表

在上一章中，我们已经大致学习了多维设计——它都包含哪些方面，以及它们是如何结合起来的。在这一章，我们开始更细致地学习有关事实表的内容。你自然会想到，事实表是多维模型中的两种类型的关系之一。（另一个是维度表，我们将在第8章学习。）

7.1 事实表的结构

图7-1显示了一个相当简单的事实表示例，它与一个客户的订购相关。这个基本的结构和使用关系模型开发的表相同——该关系由一个集合组成。事实表中的属性可以分为两种类型：维度关键码和事实。维度关键码将事实表和维度表连接起来，关于维度表我们将在下一章讨论；事实是实际需要度量的值。

和维度集合一样，事实表中的属性集合理论上讲可以为空。但是，实际上它总是至少包含两个属性：一个维度关键码和一个事实，或者有时数是两个维度关键码。

维度关键码本质上与关系模型中的外码相同，是用来连接事实表和维度表的，不过它们的主要目的在某种程度上是不同的。正如我们在第一部分看到的，关系设计的一个主要目的就是消除数据冗余以及由它造成的更新异常。

但是这样的要求并不应用在维度数据库中，因为它们很少更新。维度表的确减少了事实表中的冗余，但这却有一个副作用，它们的主要目的是提供给用户尽可能多的描述属性，用户在做数据分析时可以把它们作为选择条件。

这种“不相关冗余”的原则同样也应用于事实属性。注意，OrderNumber属性是一个事实，而不是一个维度关键码。在一个规范化的数据库中，显示在这个单一事实表中的属性将会被分为两张表，一个表描述订购本身，另一个表描述每一行的项。在一个典型的实现中，订购表中对每一条订购只有一行数据，在行条目表中每个产品都有一行数据，它是订购的一部分。CustomerID、EmployeeID和OrderDate属性将会是订购表的一部分，并且每条记录只记录一次。在示例的事实表中，它们每个属性都会为每个订购的产品重复一次，并作为订购表的一部分。（顺便注意，规范化数据库中的OrderDate将会变成事实表中的一个维度关键码。我们将在下一章详细讨论这个问题。）

这是一个典型的事实表破坏规范化表的方式，并且没有设计缺陷。记住，维度数据库中的表是很少更新的，因此在多个地方修改一个特定的数据项的问题基本不存在。这里存在一定程度的过量存储——在这个例子中，一个日期字段的大小是每个订单产品平均数目的数倍——但是当用户需要指定一个日期或者日期范围作为报表条件时，他所需作的只是一个外表连接，这样做所消耗的成本要比前面的方式小很多。

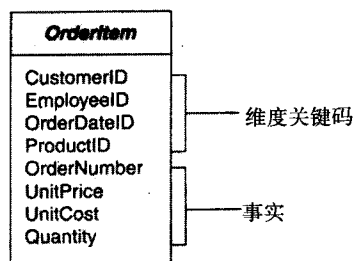


图7-1 一个简单的事实表

7.2 事实属性的特征

除了维度关键码之外，事实表还包括事实属性。（该规则存在一个例外情况，我们将在本章后面讨论它。）一个理想的事实表应当有两个特征：它必须是数值的和完全累加的。

记住维度数据库是用来作分析的。因此，用户很少会检查事实表中的每行数据。如果一个事实不是数值的——例如，它是一个日期或者一个文本字段——那么组合记录将会变得十分困难。非数值型的属性可以被统计个数或者被罗列出来，但是它们不能被累加或者求平均值。

例如，很可能只需要简单地对Quantity事实使用SQL Sum函数就能确定在10月1日售出的蓝色的大小为13的小器具的总数。但是如果返回的值是CustomerName，则只能简单地返回一个计数值：325名客户在10月1日做了该订购。理论上，你可以返回所有这325名客户的列表，但是由于我们是在一个重要的应用中，那么返回数字就和返回325 000或者甚至是325 000 000一样容易，列举事实好像没什么用处。

文本型事实是非累加事实的一个例子——该值不能被累加在一起。显然，根据定义，非数值型属性也是非累加的。这并不意味着它们永远不能被包含在事实表中，但是这确实意味着你这么做时要十分小心。在你往一个事实表中添加一个非累加属性时，请确认你不能有效地将它作为一个维度。

我从未遇到过一个非累加的数值事实，但是我不想说它们不存在。（我在这个行业太久以致不会犯这样的错误！）然而，对于数值事实是半累加的情况也并非罕见。一个半累加的事实指对于某些维度是可以累加的，但不是全部。

一个半累加事实的经典例子是一个日常账户结算。在一个指定的日子将每个账户的余额累加起来是十分合理的，只要针对账户维度累加事实数据即可，但是我的储蓄账户月末的结余并不是（不幸的）过去一个月结余的总和，那么该事实对于时间这个维度就是不可累加的。

半累加事实并不像非累加事实那样容易伪装成维度，但是它们也不是没有问题。它们不会对数据库实现的模式设计强加任何约束，但是当用户试图对一个半累加的维度进行数据合计的操作时，前端会提出警示。

关于维度数据库，我们还没有讨论的内容之一是：同规范化数据库不同，维度数据库的结构总是非常相似的。当然，维度、事实和属性的数目会有所不同，但是所有的结构都是一个事实表和围绕它的维度。

这就使得为维度数据库构建通用前端成为可能。当然，为规范化数据库构建通用前端也是可能的。Microsoft Access就是其中之一。我想说的意思是，为维度数据库构建通用前端对最终用户来说并不复杂。SQL Server Analysis Services的Analysis Manager就是一个很好的例子。

当然，这里的问题在于前端，它可以提前预测数据库的基本结构并且决定在运行时的具体模式，但是前端（或者就此而言的任何人或任何事物）不可能通过观察就能决定一个数值事实是半累加的还是全累加的。这样的信息必须被存储为元数据。尽管XML迅速发展，但元数据格式依然尚未成熟。

这意味着，最终如果你在一个维度数据库中包含一个半累加事实，那么你应当理想地使用一个客户前端，它能读取该元数据并在尝试对一个半累加维度进行求和时对用户进行提示。同时，你还需要提示用户，如果他们使用的是第三方前端工具（当然没有理由阻止他们这么做），那么用户应当自己处理这种问题。

注意，顺便要说的是当对非累加维度进行求和时，前端工具应当警示用户。我并没有说“阻止”。如果有人想根据塔什干在1953年1月和乌兹别克斯坦在2003年3月的降雨量的英寸数来划分日结算的总和，那么你能假设这个人有一个十分充足的理由这么做。

当然，这样的结果不会有任何意义，并且可以毫无顾忌地这么认为。不需要在前端阻止用户这么做。毕竟，很多年以来美国股票市场趋势的最强的预示者是女人裙子的长度。不，这没有任何意义。虽然，这曾经是个事实。但就我所知的，现在可能依然是。

7.2.1 粒度

对于一个企业运作的事实可以以不同的细节程度来度量。例如，一个销售企业的最低度量级别是一个销售订单上的每一行记录项，最高的级别是企业整体的营业净利润。事实表中的所有事实都必须在同一细节程度上，称之为事实表的**粒度**。

作为一般规则，应当以数据可能的最低级别的细节程度来构建事实表。对于大型企业来说，这可以是每个事务的级别：销售定单的行项目，或者存入某个账户或从中提取的每一笔订金，这些都是可获得信息的最小行为。这样做的原因是很明显的：你总可以对数据进行合计，但是反过来，将已经合计的数据分解成多个离散的单元是几乎不可能的。

零售操作就是这种问题的一个很好的例子。直到最近，绝大多数的零售点系统都在每天营业停止之后将总和数据发送给总部：“我们卖了16个绿色的小部件，36个蓝色的小部件，42个大型工具，以及一只在一个棵梨树上的鸬鹚”。这些都是很有用的信息，但是至此它们都不能再被细化了。

如果可用的信息只有每日销售的产品，那么你就不能实现所谓的“市场指数分析”，回答诸如“有多少人既买了牛奶也买了面包？”的问题对市场类型的分析十分重要。

顺便提一下，这个特定问题的答案就是超市应当将牛奶和面包分开放置，并且离得尽可能远的放置的原因。既然大多数人买了其中一个也会去买另一个，那么商店将它们分开放置就可以使得顾客路过尽可能多的其他产品。这个理论就是，你看得越多，你就越有冲动购买。（老实说，在我为一个澳大利亚最大的连锁超市做这些项目之前，超市购物要有趣得多。）

但是回到主题。这个原则就是细节程度越高，分析的方式就越多，因而也就越有用。但是，你只能获取已经存在的细节。在我们的例子中，市场指数分析所需要的信息就是不存在的——很不幸，但是没有协商的余地。这看上去好像有点陈腐，但是如果你对此做得足够多，我可以保证在某种程度上，对某些人来说不够清楚的一些市场行为会由你个人负责实施。

当你从多个系统中抽取事实时，粒度也会成为问题，这也是一个常见的情况。例如，假定你在构建一个事实表，其粒度是单个的订购行条目，并且销售价格条目和订购折扣条目都来自于订单输入系统。到目前为止是好的，但是在其中包含每个条目的成本会不会更实用呢？这样分析人员就能实施净利润的计算。

这是个很好的想法，但是记住我的话：你永远不可能在条目级别获取成本的完整描述。毕竟，一个完整的描述还要包括类似CEO的花费账目，而且即使该项信息可以获得，也不可能将其有意义地分配到系统中。

给定一个完整的描述是不可能的，你不得不确定（当然需要和你的最终用户商量）一个合理的描述。大多数的销售公司为了账目的目的都有某些确定产品销售成本（Cost of Goods Sold, COGS）的方法，并且这也是一个很好的开始点。但是你很可能发现产品销售成本是在

订购级别度量的，所以如果只是简单地通过使用产品销售成本总和除以条目数的方法来分配成本，将是不可行的。

当然，有时候你会比较幸运。我曾经为一家公司工作，它就是把销售价格的50%作为COGS。他们不可能运行长久，但为他们构建一个简单（廉价）的系统，的确是很好的调剂。

你可能还会发现最终用户想要包括（和分配）那些在COGS之外的成本——例如运输和广告成本。作为一个分析人员，你必须很清楚你的位置：为了在维度数据库中包括一个事实，在某种粒度上分配该事实需要所有感兴趣的参与者达成共识才行。如果它得不到同意，那么就不包括它，讨论结束。

建议你尽可能远离分配讨论。我十分惊讶地发现人们对这些事情都十分激动。我曾经主持过这样一个会议，在会上讨论关于一个广告活动的分期付款问题，销售副主管威胁说要把市场部副主管从15层楼的窗户里扔出去。（这是我职业生涯中唯一的一次将咖啡时间变成了一场自卫。）幸运的是，争论者不是经常走向这样的极端，但是也要准备好它偶然发生，所以我们说是“热烈讨论”嘛。

一旦用户在分配规则上达成了协议（或者让事实数据不那么棘手），接下来重要的是将这些规则完全存档，可以是元数据的形式，也可以是（最好是）作为关联维度的一个属性。

如果你参与了多个数据集市的工作，那么这一点尤为重要。记住我们将一个数据集市定义为一个维度数据库，并且同时使用多个维度数据库也是很经常的事情。为了使工作顺利，必须在整个系统中始终如一的使用相同的规则，并且只有当它们是在自己的数据集市内部存档时，才会使用到这个原则。

7.2.2 事实表的类型

自此我们都在集中关注最常见的事实表类型，它是对一个事务建模。通常来说，我们把这种结构称作一个**事务表**。但是这里还存在一些你需要注意的其他重要类型的事实表。

前面我们提到了日常账户结算是半累加事实的一个例子。那些描述余额的事实几乎总是第二种重要的事实表类型——**快照表**的一部分。正如你所期望的那样，快照表描述的是企业状态的周期性快照。

当你需要对某个值随时间变化产生的差异作度量的时候，这种类型的表就能发挥作用了。除了账户结算外，比如你还可以使用一张快照表来度量库存水准、股票抛售价格、学生注册人数，甚至一个房间的温度。

请注意，并不是所有这些事实都会受到事务的影响——库存水准会，而温度不会。如果它们会受到事务的影响，那么习惯上数据仓库应当同时包含一张快照表和一张事务表。

当然，你可以通过初始余额和记录的事务来计算余额：这正是记账系统所做的。维护余额快照需要使用大量的内存，但是计算可能会是一个复杂和耗时的过程，它可能需要数以万计的个别操作。请记住，存储是廉价的；时间是宝贵的。

一种特殊类型的事务/快照对被用来对自然增长的业务进行建模，例如杂志、保险、有线电视的业务，或者某种金融事务，比如贷款。通常这些行业都是预先收到他们的服务报酬，但是他们实际上在服务（或产品）交付前是没有增加收益的。

举个例子，比如我订阅一份杂志，十二期共\$24，出版公司在我订阅的时候收到了\$24，但是却只能在他们每次给我寄送杂志时才可以记下\$2的收益。直到这时候为止，该订阅收益

只能算作债务，而不是资产。（是不是很有趣？但我不这么认为。）

在类似这种情况中，你肯定不会尝试在运行的时候进行收入的计算。即便你是对一个行数相对较少的数据集进行操作，该计算也会需要日期计算，这是相当慢并且容易出错的过程。此外，如果你是在一家跨国公司工作，那么问题将变得更加严重。如果客户是在加利福尼亚，但是家庭办公室是在澳大利亚的新南威尔士，这个城市位于国际日期变更线（International Date Line）的远端，那么该客户的收入是在加利福尼亚的某月的第一天增加还是在新南威尔士的前一个月的最后一天增加呢？最好是让会计部门来处理这些事务，毕竟，这是他们的工作。

除了快照表和事务表之外，还有最后一种你可能遇到的事实表类型：**覆盖表**。有时候仅是包含一组特别维度标识的一行数据就能告知你想要知道的一切信息，或甚至是你能够知道的一切信息，而不需要其他的事实。

覆盖表有时候也被称为“无事实数据的事实表”，它通常被用来表明一些已发生的事情。各种维度标识完全定义了该事件，并且该行数据的存在就足够来表明该事件发生了。

例如，如图7-2所示的事实表就可以用来对课堂出勤情况进行建模。

一些分析人员在规范的结构中添加一个额外的布尔事实——总是为True，如图7-3所示。这里的理论依据是使得用户更容易理解和操作该表。

从个人观点来说，我并不赞成这样的做法。首先，布尔值既不是数值的也不是可累加的。事实上，由于它完全不能累加，因此使得它成为一个相当难看的事实数据。第二，一个True值的存在暗示着存在False的可能，但这是不可能的。记住事实表总是稀疏的。绝不要在其中添加表明一些未发生的事情的数据行。这就好像是证明一个否定的命题，这种方式是十分不明智的。

但是，不得不承认，这个论断是主观的，并且在某种程度上是出于审美的考虑。最好的办法应该是与你的最终用户商讨。毕竟，重要的是该模式对他们有意义，而不是对你。但是我可以警告你，你会发现很难解释这个问题。

7.2.3 异类事实

在第3章中，我们谈论了在规范化的数据库中，子类实体作为描述那些既要作为常规类型处理又要被看成特殊类型的实体的建模方法。当然，在维度数据库中也存在同样的问题，并且解决方法是类似的，但并不完全相同。通常，在维度数据库设计中，这些被称作**异类事实**。

图7-4是摘自第3章的示例，它显示了在一个规范化数据库中的一般方法。Products实体包含每个产品的共同的属性，而每个个别产品类型实体——Beverages、Condiments等等——包含该类型的特殊属性。在Product级别包含一个ProductType属性可以使得处理任何特定的记录或者一组记录像处理一个普通产品或者一个特定类别的实例一样简单。

Attendance	
DateID	
ClassID	
TeacherID	
StudentID	

图7-2 一个对课堂出勤情况建模的覆盖表

Attendance	
DateID	
ClassID	
TeacherID	
StudentID	
Present	

图7-3 带有一个人为布尔事实的覆盖表

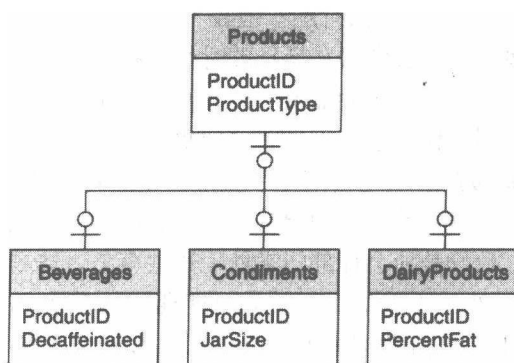


图7-4 来自规范化数据库的一个子类实体

规范化数据库支持的应用程序通常就以这种方式同时在两个级别上对待实体。例如，一个产品细节信息列表需要为Products表中的每个记录显示其一般的属性以及类别相关的字段。这种要求在维度数据库中比较少见。实际上，用户或者是在一般级别上分析数据，或者是在类别一级上分析数据，而不会同时在两个级别上分析数据。继续我们的产品例子，分析人员想要通过产品类型来分析所有产品的销售情况，但是该分析不需要任何特定产品的细节信息。同样地，一个想要分析Condiments类别的产品销售情况的分析人员只会关注该类别，而不需要Beverages或者DairyProducts的细节信息。

由于这种使用方式的不同，最好避免表的连接，这时可以为一般类型和每个类别分别创建一个不同的事实表，如图7-5所示。

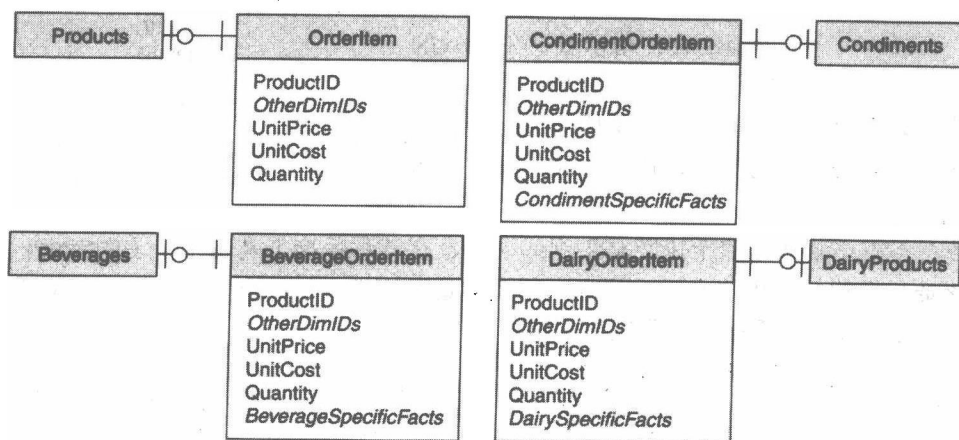


图7-5 表达异类事实的一组事实表

在这里需要注意两点：第一，在分析中所使用的事实数据会在一般和特定类别的事实表中重复出现。复制这些事实数据需要一些额外的存储空间，但是它会避免不必要的表连接。这种方式一般会大幅度提高系统性能。

第二个需要注意的事项是存在个别的表，不仅用于事实数据，也用于维度数据。这不是绝对必要的。有时候，并不存在某个类别维度特有的维度属性，而且所有的维度属性都能合理的存在于一般级别。在这种情况下，维度间就不存在重复的值了。

但是，通常每个类别都有一些不与其他类别共享的维度属性。在这种情形下，多个表会更有效些，因为它们可以避免大量的空维度记录。

无论你怎么处理，都要试图避免在运行时定义属性，比如类似 “If ProductType = ‘Beverage’, then Att1 = Decaffeinated; if ProductType = ‘Condiment’ then Att1 = JarSize” 的定义。我偶尔在事务性的系统中会看到这种间接的模式定义——这些系统都是由相当聪明的人设计和维护的。通常，这些系统会在一个致命错误中崩溃，比如需求的改变或者原创设计人员的离开。

类似这种间接的模式不可能在一个分析系统中维持很久。记住星型模式的一个主要目的就是易于最终用户的理解。一个间接的模式永远不会满足这个目标。

7.3 小结

在这一章中，我们探讨了维度数据库中的事实表的结构。我们看到一个事实表由两种类型的属性组成：维度关键码和事实。维度关键码在结构和功能上都类似于关系模型中的外码，它们用来连接维度表和事实表的数据行。

事实表中的事实数据代表了由维度数据的特定组合所度量的值。理想的事实有两个特征：它是数值的，并且是完全累加的。非数值的事实并不一定是错误的，但是当用它来表述时，需要格外注意它确实确实不是一个伪装的维度。

由定义可知，非数值的事实是不可累加的。它们可以被计数和罗列，但是不能以任何方式求和。完全累加的事实可以被计数和罗列，也可以对任何维度进行求和。半累加事实是数值的，但是只能通过某些维度求和，而不是所有的。

一个事实表的粒度就是获取事实的细节程度。你应当始终使用尽可能合适的粒度，因为细节程度越高，数据分析的方式就越多。但是，应当注意不是所有的源数据都能在同一粒度上可用，并且将这些高级别的事实分配到较低级别上的方式是一个有争议的过程。

有三种主要类型的事实表。最常见的是事务表，它是对每个事务进行建模的，比如一个销售订单的行项目。事务表经常和快照表结合起来，快照表可以度量在某个特定时间点的数值。最后一种类型的事实表是覆盖表，它实际上可能不包含任何事实属性。覆盖表最常用来记录已经发生的事件。

最后，我们讨论了如何处理异类事实，它类似于在关系模型中对子类实体的处理过程，异类事实可以使用多个事实表/维度表来处理，一个对是处在一般级别，一个对是为每种类别。但是，与子类不同的是，异类事实在每个特定类别的表中都重复核心事实。这避免了大表之间的连接。

第8章 维 度 表

在上一章，我们探讨了事实表，它是每个星型架构的核心部分。这一章，我们将讨论维度数据库中的另一种类型的表——维度表，当然，多维模型就是由此得名的。

在某些方面，维度表类似于规范化数据库中常见的查找表，但是尽管它们的外表相似，但维度表却是服务于完全不同的目的。查找表是为了减少重复数据和保证一致性。维度表表达的是将被度量的数据的特征——它处于一个度量使用的一组维度值的相互关联处。虽然维度表从某一方面来讲能够减小事实表的大小以及它包含的重复数据的数量，但它们的主要目的是方便分析。通过这种方式，分析人员可以在大量数据中进行钻取和挖掘。

8.1 维度表的结构

一个典型的维度表比一个事实表要宽泛，实际上要比规范化数据库中的典型表都宽泛。大多数事实表一般不超过3个或4个事实数据，而规范化的数据库大约会同时查阅超过20或30个的属性值，而包含50个属性的维度表是很常见的，超过100个属性的也不罕见。事实上，这是合理推翻很多关系引擎强加的255个字段的限制的一种情况。

可能最好的例子就是几乎每个事实表都会含有时间维度，比如一个事务或者一个快照行的数据。在源（规范的）数据库中，日期字段通常都是一个单一的SQL日期。在一个维度数据库中，这个单一的字段可以被一个指向包含众多属性的维度表的链接来代替。表8-1显示了一个零售应用程序的典型例子。

表8-1 一个典型的时间维度

Attribute	Example
TimeID	8302984
FullDate	14 June 2004
DayOfWeek	"Wednesday"
DayInMonth	23
DayInYear	173
CalendarYear	2004
Quarter	"Second"
FiscalYear	0304
FiscalPeriod	"2Q"
HolidayFlag	True
HolidayName	"Flag Day"
WeekdayFlag	True
WeekendFlag	False
BeginningOfMonth	False
MidMonth	True
LastDayInMonth	False
Season	"Spring"
EventFlag	False
EventName	" "

这里有几个注意事项。第一，如果花点时间查看这些属性，你会发现它们都试图给分析人员可能提出的每一种问题提供直接的支持。我们在周一或者周三会卖出更多的土豆吗？足球比赛会影响咖啡的销售吗？哪些产品在月初和月末（标准的薪水日）的销售会达到最高点？

第二，实际上，SQL日期已经被TimeID码所替换，很可能它在时间维度中远没有在产品列表中重要，但是将任何从源数据中提取的关键码换成无意义的替代码是一种很好的方法。

指定替代码的主要是为了在维度数据库中避免源系统中关键码值的改变。不论源系统的管理员有多么坚持关键码不允许改变，它们都会发生改变。只要有足够的时间，源系统就会改变建立关键码的方式，可能是因为事务系统被替换了，或者因为公司与其他企业合并了，又或者仅仅是因为管理员心情不好。（不要笑，我见过这类事情。）

或者，更糟糕的是，源系统仍旧使用原来的关键码。这么做对一个事务系统来说是完全合法的事，它不关心企业的历史信息。但是重用关键码对一个维度数据库来说是破坏性的做法，因为维度数据库跟踪历史数据。PL-3957政策指的是由Peter Levinson主持的当前管理者的决议，还是指由Acme公司在1957年提出的Public Liability政策？这样的问题在事务系统中永远不会出现，但是它可能存在于维度数据库中。

在维度数据库中使用替代码的第二个好处就是你可以使用一个更小更合适的随机数，而不是从源系统中获得的数据类型。这通常不会有问题，但是当你陷入与那些不了解维度数据库如何工作的人的争论时（这类事情经常会发生的），这可以成为你有利的理由。

顺便说一下，请注意源SQL日期在维度表中以FullDate属性的形式存储。很显然，你始终都需要保留原值。你可能还想要存储事务系统的名字，它最初将该值设为一个主码。这在系统改变或者该关键码重用时会十分有用。（我们将在本章后面详细讨论有关维度值改变的问题。）

在表8-1中，另一个需要注意的事情是很多属性是（或至少可能是）基于一个或多个其他属性的计算得到的。例如，Month、DayInWeek和DayInMonth很容易从完整的日期中计算出来。当然，如果这是一个规范化的数据库，那么这必然与范式相冲突。但是它不是，所以该规则不适用。

记住维度表中的属性是被用来限制查询的。在实际应用中，这通常意味着一些可用的值会为用户显示在一个组合框中。现在，如果我们以Month属性作为例子，那么在一个“非规范化”的维度模式下

```
SELECT DISTINCT Month FROM Time;
```

和在一个规范化数据库模式下

```
SELECT DISTINCT MonthName
FROM CalendarNames
INNER JOIN NormalizedDimension
ON CalendarNames.MonthNumber = Month(SQLDate);
```

操作该组合框是有所不同的。甚至不用做正式的性能测试，我敢打赌第一种形式的语句会比第二种要快很多。当你操作多个维度时，这种性能优势将会更加明显，而你所做的典型分析通常都需要多个维度。记住：存储空间是廉价的，而时间是昂贵的。

一个类似的原则解释了WeekdayFlag和WeekendFlag属性对之间存在的相互排斥。的确，使用

```
WeekdayFlag <> False
```

和

```
WeekendFlag = True
```

将会返回相同的数据行集。但是后者对于用户来说要更容易理解和操作。

对于表8-1中的有关属性，需要注意的下一件事情是不存在缩写。我最近进行了一次网上订购并且收到一封电子邮件，确认我订购一个FB11 T WHi。由于确认信来得十分快并且我只订购了一样东西，所以我十分清楚FB11 T WHi代表的是white, twin-size, feather bed。但是如果一个月后问我，我肯定就没有任何印象了。

这里的关键问题不是该公司发送有问题的确认信（虽然它们的确是），也不是我的床垫这些天更舒适了（虽然它的确是）。关键问题是FB11 T WHi很显然是公司产品主文件中的“描述”字段的内容（产品编号甚至更糟糕：FB11-378095X）。这很可能对那些整天做订购、包装和运输的人员十分有意义，但是作为在维度数据库中一个产品的描述，它是很不恰当的。

作为开始，你不能期望任何刚刚接触数据仓库的人就能记住“WHi”代表“White”，特别是由于它很可能在表的其他地方代表“Whip-stitched”。在我们的表中，是“Wednesday”而不是“Weds”，或者，是“White”而不是“WHi”，请将整个单词拼写完整，而不要让用户去猜。

除了这个问题之外，请注意在“FB11 T WHi”中存在着多个信息。它告诉我们该产品是皮质的床垫，它的类型是11（猜测它可能是第11个进入系统的皮质床垫，但从表面观察无从得知），它是双人床尺寸的，并且它是白色的。现在，如果这些信息能够在不同的字段中作为选择条件使用，那么将所有这些信息都存入一个描述字段中也是可取的，不论对产品的控制模式是怎样的，该维度必须要与表8-2所示的样子类似。

表8-2 一个来自产品描述示例的产品维度

Attribute	Example
ProductID	10293810
ProductMasterID	"FB11-378095X"
Category	"Bedding"
Product	"Featherbed"
StyleNumber	11
Size	"Twin"
Color	"White"
Description	"Twin White Style 11 Featherbed"

不幸的是，不论产品的主文件是怎样的，将它们转变成这种美观、整洁和易于理解的维度都不是一个简单的过程，这个过程称作清洗（scrubbing）。它通常需要人工干预，并且可能引起很多行政上的争议，正如我们在上一章中看到的分配成本的问题一样。

你可能将要面临的最大问题是，首先要确定谁“拥有”这个产品的主导权，或者当用户第一次使用时要查看的维度。在一个产品列表中，可能存在多个组的人员声明拥有权：生产部门、销售部门、市场部门，甚至运输部门，要么不得不让他们都参与到清洗过程，要么求助上级主管来解决这个行政问题。

或者，你可能处理一个“灰姑娘”的维度，它参与每个人的工作，但是没有被任何人声明。关于一个产品主导权的问题一点也不奇怪，特别是在遇到一个暧昧的微笑和对于使用产

品列表的多个系统的讨论时。后一种情况对于维度数据库项目实际上是件好事。它意味着维度数据库可以获得产品主导权（或者其他维度）的拥有权，一旦将它清洗过后，其他系统就免于被破坏和捣乱。至少，理论上是这样的。

8.2 雪花化

还有最后一件需要注意的事情：在表8-1和表8-2中所示的维度的结构不是规范化的。在维度中的一对多联系并没有像规范化数据库中那样抽取到不同的表中。

例如，由表8-2和图8-1所描述的维度中，很有可能在Category“Bedding”中存在多个Products，因为在这两个属性之间存在一个一对多的关系。在Product和Style之间也有一个一对多联系，并且在Style和Size以及Style和Color之间还有两个一对多联系。

在规范化的数据库会要求将这些一对多联系分别构建为不同的表，如图8-2所示。这种架构会更有效地使用空间，并且减少数据输入错误的可能。

Products
ProductID
ProductMasterID
Category
Product
StyleNumber
Size
Color
Description

图8-1 一个Product维度

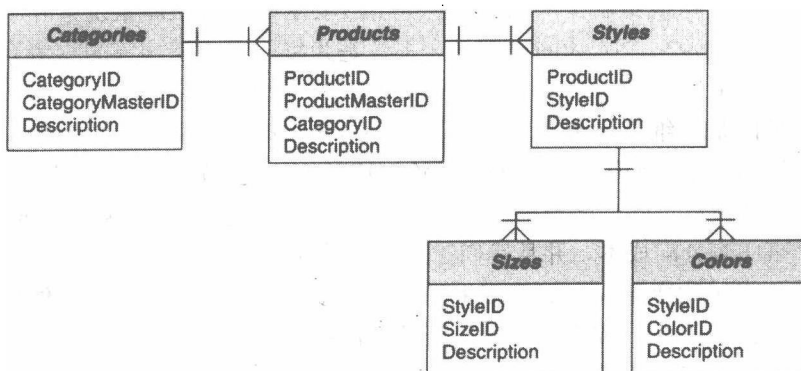


图8-2 图8-1所示的Product维度的规范化版本

但是正如我不断重复地，这些并不应用在维度数据库中。（我意识到我十分唠叨，但规范化的强烈要求的确是一个很难破除的习惯。）一个维度数据库的模式就是使得它易于分析人员理解和浏览。在维度设计中，规范化一个维度称作雪花化，但它并不能达到这样的目的。

有一种情况是合适雪花化的：当维度和事实表之间存在一个多对多联系的时候。这种情况在维度数据库中很少见。大多数的多对多联系是存在于维度之间的，并且在这种情况下，事实表作为连接表就能解决这个问题。

但是维度和事实表之间的多对多联系偶尔也会发生，并且如同关系数据库的任何多对多联系一样，它们必须使用一个连接表来解决。这种情况的一个经典例子是一个描述到医生办公室就诊的事实表，以及一个描述诊断的维度表。显然，一个医生能在一次就诊中诊断多个问题，因此在维度和事实表之间的多对多联系就必须构建为如图8-3所示的形式。

这种情况与规范化数据库中的模式非常类似，但是请注意，该连接表包含一个叫做

“Weight”的属性，这在规范化数据库中是不常用的。该Weight属性是一个数字值，在Diagnoses维度被用来限定一个查询的时候，Weight就用来指定数值的事实数据。

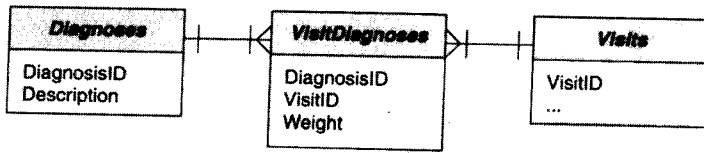


图8-3 诊断维度和就诊事实表之间的多对多联系

每次就诊的Weight值的总和必须等于100。因此，如果Ms. Bloggs由于伤风来Dr.Jones这就诊，但是当她就诊时，医生还调整了她血压的药物治疗，那么将会有两条记录添加到VisitDiagnoses表中，例如，赋予伤风80%的权值，而给高血压20%的权值。

之后，如果一名分析者询问所有有高血压诊断的就诊的总费用（即，Diagnoses维用于限制这个查询），那么Ms. Bloggs就诊的20%的费用将会包括在结果中。另一方面，如果要求查询Ms. Bloggs属于的其他某个组——比如20岁以下的女性——的费用总和，那么花费的100%都应该包含在结果中。

现在这种方法的问题很明显了：在极端情况下所要求的权值在原系统中并不存在。如果它不存在，你不得不生成一个人造的权值（大多数是为每一项分配相等的权值）或者修改原系统。在我们的例子中，要求医师在他们的表格中添加分配的权值。

可能不太明显的是，维度和事实表之间的多对多联系有时候能够通过调整事实表的粒度来消除。在我们的示例中，如果我们改变从就诊到诊断的粒度，这个多对多联系就不存在了。

不幸的是，这并没有消除分配值的问题，它只是转移了。虽然不需要对维度组分配权值，但是需要对一个事实表分配成本，并且我们在上一章所讨论的问题依旧存在。

但是，这并不表示改变事实表的粒度来消除多对多联系就不值得考虑。偶尔以不同的方式考虑问题的确可以解决它，或者至少可以成为解决棘手的分配问题的一种提示。虽然只是偶尔如此，不过它依然值得一试。

当设计一个传统的规范化数据库模式时，这个设计过程相对来说总是比较简单的，只需要确定数据的含义，然后在其结构上直接应用逻辑规则即可。但是在设计一个维度数据库时，尽管它有简单、可预料的物理结构，但它通常是一个复杂的过程，包括要根据可用的数据的粒度来平衡事实表的粒度、并且开发可行的且政策上可接受的分配公式。我很想说这就像网络的复杂性一样，但是可能这样的论调过于抽象而导致没什么联系。

8.3 改变维度

自此在对维度表的讨论中，我们基本上都将它们视为静态的。我们的确说过使用替代码来使得数据免于改变，但是我们并没有进一步探讨如果发生了改变该如何处理。当然，它们的确是可能发生的。

当维度行中属性的值发生改变的时候，你有三种基本的选择：

- 可以使用新的值覆盖旧值（会丢失历史数据）。
- 可以添加新的一行（保持了所有的历史信息，但使得设计变复杂了）。
- 可以同时保留最初的和当前的值（丢失了中间值）。

对于处理维度的改变，覆盖旧值是最简单的方法。这种方法通常用在事务数据库中，它们通常只关心实体的当前状态。但是维度数据库感兴趣的是历史的趋势，在这种前提下覆盖旧值必然会带来问题。

比如，作为一位住在荷兰阿姆斯特丹（我那时在那）的女性，我于2000年3月被添加到了一个Customer维中，并且价值\$1 175.32的销售事务也被添加到了事实表中，并指定了我的CutomerID。在2002年的4月，我的地址迁到了新墨西哥的圣达菲，而且我进行了一次价值\$1 053.97的交易。

现在，订购输入系统并不关心我曾经居住在阿姆斯特丹，它只想知道将我的账单寄往哪里。但是如果维度数据库中的值被覆盖，而当分析人员在2004年询问过去5年中所有新墨西哥居民的订单的总和的时候，这个系统则会错误地将我在阿姆斯特丹所花的\$1 175.32也包括进来了。

由于维度数据库关心的是历史数值，你不能简单地将它们丢弃或者覆盖。使用这种技术的唯一情况就是原始记录值是错误的——比如我并没有首先在阿姆斯特丹居住过。当然，不论原系统的改变是一个错误的修订还是一次改动，这个问题都会存在，因为大多数事务系统不会记录这些信息。

我担心解决这个问题又会是一个策略上的事情。虽然可能性不大，但还是有可能要改变原系统。偶尔的管理要和恰当的推测同步而行——在某个时间段中的任何改变，例如7天，均视为一次修订。这之后系统将它看成是一次实实在在的改变。大多数时候，不得不人为的改变的状态视为清理过程的一部分。

假定你已经确定这次改动确实是一次改变而不只是修订，那么就必须选择另外两种方法中的一种来处理它。你的选择应当依据（通常是这样的）数据的含义以及使用方式。顺便提一下，这两个选择并不是相互排斥的。你可能决定在某些字段改变的时候创建一条新记录，而在同一个维度中，对其他改变使用一对原始/当前值。

在改变发生的时候创建一个新的记录行是最有效的解决方法，但是它要求该维度的标识符同时包含一个一般标识符和一个序列号。例如，为了保证现在居住在新墨西哥的Rebecca M. Riordan和之前居住在阿姆斯特丹的Rebecca M. Riordan是同一个人，该维度应当分配一个一般CutomerID，例如35972，并指定序列号1给阿姆斯特丹的地址，指定2给新墨西哥的地址。任何时候，我的完整的维度标识符就应当是35972-x，x则表示当前的序列号。

为了更容易提取最新的数值，一些分析人员使用一个任意数，比如999来标明当前的值。在这种情况下，系统就需要在初始的时候指定999给阿姆斯特丹的地址，并且当我搬家之后，则应当将阿姆斯特丹指定为1，而将999分配给新墨西哥。如果你确信这个策略不会被破坏的话，这种方法的确能够使得“当前值”的分析简单很多。

除了处理改变之外，有时候还需要知道改变发生的时间。要做到这一点，有两种方法。你可在维度表中存储有效的日期，它对跟踪某个维度实体的历史以及针对维度表本身实施查询都十分有用。或者你可以在事实表中同时存储实体的一般标识和当前序列号，这样能够简单无误地将事实按时间划分开来。

大多数作者推荐后一种方法，但是实际上它们并不矛盾，我认为包含两者会更安全些，同时在维度表中存储一个有效日期，并在事实表中存储一个序列号。请记住，空间总是廉价的……

最后一种处理值改变的方法是当你不需要过多地跟踪历史数据，而只需要同时记录当前和原始（或者之前）值的时候使用。例如，这种结构可以简单地回答诸如“我们有多少新墨西哥的顾客是从阿姆斯特丹迁移过来的？”之类的问题。现在，这在一个销售环境的确不是个恰当的例子，但是它对于一个对人口统计趋势感兴趣的系统——比如疾病控制系统来说就十分有意义了。

在一个销售环境下，当有效日期有些失真，并且在某段时间内认为没有变化发生是合理的时候，这种技术被称作所谓的“软”改变。典型的例子就是产品配方。在七月初生产YaYa Gel产品可能使用的是配方B，但是很可能在仓库、运输部门以及商店里依然在出售使用配方A的YaYa Gel。在很多企业中都存在一个不可预测的时期，此时YaYa Gel销售的可能是任意一种配方。

下次你在超市的时候，可以查看一下那些成分列表存在这样的警示语“可能包含花生或者花生的副产品”的产品编号。这些产品的配方每一批都稍微有所改变，植物油或者花生油的选择是依据商品定价来确定的。这是一些你可能使用Current/Original或者Current/Previous属性来处理改变的情况。

不过，这些情形是很少见的。事实上，你更经常遇到的问题是“我是否关心”而不是“我以何种方式关心”，并且当你面对后者时，依我的经验来看，在新记录行和当前/之前属性对这两种技术之间的选择实际上是十分明显的。

8.4 小结

本章我们讨论了维度表。我们介绍了它们的基本结构，并且讨论了为了使维度数据库免受原数据模式的改变的影响，最好的方法是给所有的维度添加一个人造码。

我们还讨论了雪花维度模式，并且看到了如同在规范化数据库中分解一个关系一样，它在类似的情况下是十分有用的。

最后，我们探讨了维度改变的问题，并且介绍了处理它们的两种技术：要么添加一条新行，或者使用一个事实对来表达当前和以前的值。

自此我们就结束了维度数据库理论的介绍。是不是并不像你想象的那么困难呢？在第三部分，我们将转向讨论设计数据库系统的过程。

第三部分 设计数据库系统

第9章 设计过程

在第一部分和第二部分，我们关注了关系数据库和维度数据库的设计原则。但是数据的结构只是数据库系统的一个组件——当然它是一个很关键的组件，不过也仅仅是一个组件。从这一部分开始，我们将关注有关设计数据库系统的其他方面的内容。

在本部分，我们将讨论涉及数据库系统分析和设计的大部分工作，包括定义系统参数和工作过程，概念数据库模型以及数据库模式。由于用户界面的设计是一个相当复杂的话题，所以我们将在第四部分讨论它们。

我们首先介绍的是分析和设计数据库系统，有关实现的内容超出了本书的范畴。但是分析和设计不能脱离其余的过程独立存在，所以我们先大致讨论有关生命周期的内容。

9.1 生命周期模型

曾经有一段时间，系统分析人员使用名为**瀑布模型**的范例作为开发过程的指导。关于这个模型存在多个版本。一个合理和简单的版本如图9-1所示。

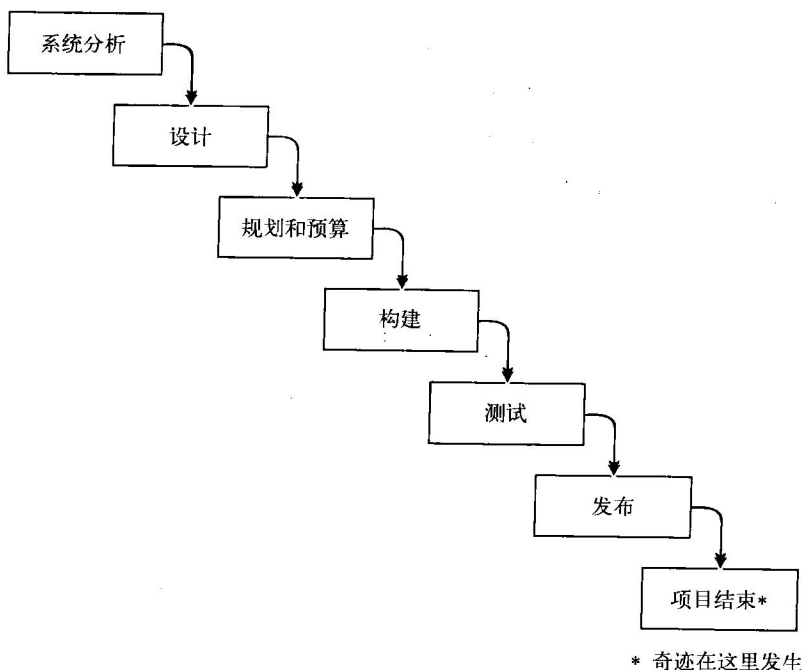


图9-1 瀑布模型

该过程从系统分析开始，有时称作需求分析，因为它主要关心的是企业和用户要求系统做什么。一旦系统分析完成并得到了认可，整个系统才进入详细设计阶段。这个阶段跟在计划和预算之后，接着是构建、测试和发布整个系统。至少理论上是这样的。

瀑布模型很受欢迎。每一项活动都在下一项开始前完成并得到认可，并且该模型可以有效地控制预算、人员配置和时间。客户会十分愿意对时间和预算做一个瀑布项目。

当然，这种方式存在的问题是现实总是很难完全切合要求。该模型假定完成一项任务所需的所有信息在实施该任务时都是可用的，并且不允许在过程中加入新的信息。除了一些十分小的系统（那些你只用花一个星期的时间就能构建好的系统）之外，这种理想的情况是不存在的。

瀑布模型还不允许在项目进行中变更业务需求。要求一个系统在项目起初直至两三年的开发过程结束都满足业务要求是绝对不可能的。如果你发布了一个无用的系统，即使它是按时完成并在预算内的，你的客户也会对你不满意。

但是，理解瀑布模型中的活动标识是十分有用的。事实上，在开发项目中遗漏它们其中的任何一个都是不允许的。该模型的问题是它是线性的，它假定每一阶段一旦完成后就不再需要重新检查。

为了解决瀑布模型的这些问题，人们提出了几种其他的生命周期模型。**螺旋模型**采用的是多个反复的瀑布方式，每一个都扩展了前一种的范围，如图9-2所示。

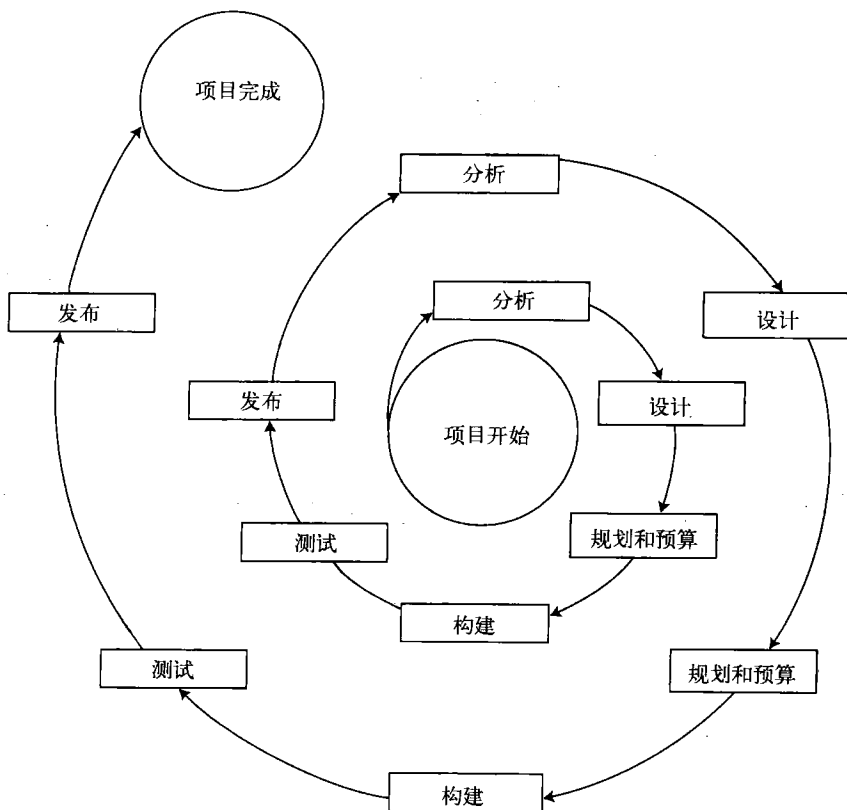


图9-2 螺旋模型

螺旋模型的问题是，当严格应用该模型的时候，项目的整体情况只有在开发的后期才能考虑，并且很有可能后期的迭代会破坏之前的工作。对我来说这通常是一种提高预算却打击开发人员的方法。这种情况对数据库项目是十分危险的，扩展业务范围会改变数据的语义，并要求改变数据库的模式，而数据库模式的改变可能会导致整个系统无法预知的改变。

在大型系统中以及在我个人的工作中，我倾向于使用被称为**增量开发**或者**进化开发**的模型，如图9-3所示。

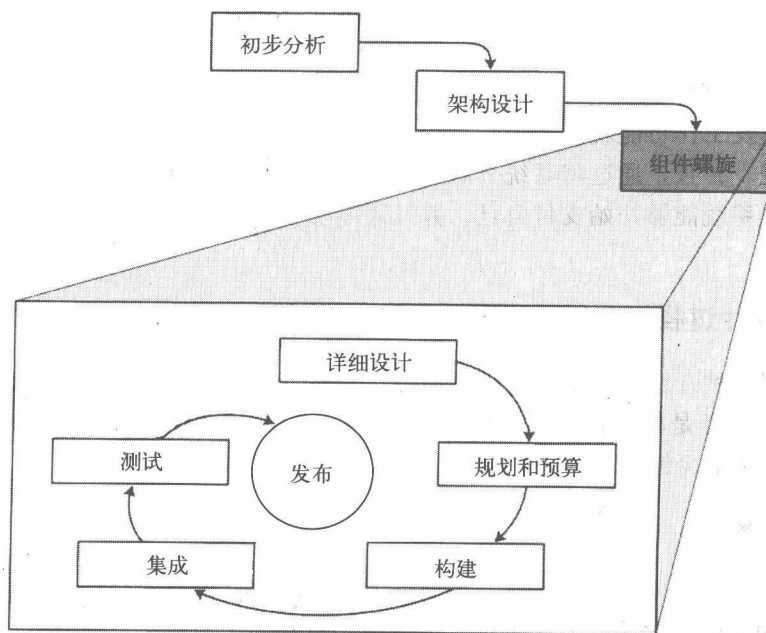


图9-3 增量开发模型

在这个模型中，它有很多方面都与螺旋模型不同。它最初的分析是针对整个系统的，而不仅仅是某一部分。接着是架构设计，然后是整体分析。架构设计的一个目标是定义单个组件，它们或多或少都可以独立实现，另一个目标是描述这些组件之间的交互性和相关性。接下来对每个组件的详细设计和实现可以使用任何最合适的模型。我们在这个阶段选用的是螺旋模型，如图9-3所示，因为它在设计和实现过程中有更大的灵活性。

注意，这里的螺旋模型包含一项额外的工作：集成。当然，组件集成在螺旋模型中是隐式的，但是以我的经验来看，该任务如果使用增量开发方法会变得更复杂。这也是我为什么在组件开发阶段用螺旋模型的原因之一。将一个组件的详细设计拖延至开发它之前，使你能够调整之前组件集成过程中的细节，并且如果幸运的话，还能够避免你在集成过程中可能遇到的问题。

增量开发模型的问题是它假定任何大型的系统都可以被划分成不同的组件，但这并不是对所有系统都适用的。同时它还可能导致很多“框架”代码。例如，有一个数据输入界面，它要调用一个COM组件来对客户代码进行查找，并且当找到匹配项的时候便采取相应的措施。但是，该COM组件尚未构建完毕。开发小组不得不先构建一个虚拟组件以使该调用不出错。复杂的系统可能包含大量类似这种框架的代码。

除此之外，外部组件很可能永远不会被实现；有可能是由于预算达不到，或者你之后觉得这个组件并不实用。如果你有计划这种可能性的话，那么你可能会发布那些只是确保其他组件不会产生问题的组件。这样的情况是不利的，当然你也不愿意就此对维护人员进行解释。

因为分析和架构设计是在项目开始的时候实施的，因此它们就有可能会过时，这可能会使你回想起瀑布模型的一个主要弊端。鉴于这一点，在开始每个组件的详细设计之前，重新检查这两步是很重要的——特别是需求分析，因为它最有可能发生改变。通过修改尚未开发的组件或者通过调整新组件开发的次序来适应需求的变动通常会十分让人高兴，这样就不会导致之前的开发工作无效。

尽管有这些风险，增量开发模型也有不少优点。因为系统的“概貌”在开始的时候就确定了，无用的开发工作可能会减小到最少。因为大项目被分解为小的组件，各个组件项目会变得容易管理。并且，通过将系统分解为不同的组件，你可以将一些核心功能提前提交给客户。这就使得系统能够开始支付自己，并且还能够获得用户的信息以反馈给后续的开发工作的机制。

9.2 数据库设计过程

不论选择哪一种开发模型，都必须实施分析和设计工作。不论你是顺序还是迭代地实施它们，不论你的任务是整个系统还是单一一个组件，不论你的技术是正式的还是非正式的，每一个项目都必须至少包括以下步骤。

9.2.1 定义系统参数

理想情况下，每个项目都应当以一个清晰的定义开始：你要达到什么目的、为什么你要达到它、以及如何评估你的成败。不过，大多数项目在开始前都没有这些定义，因此这就是设计过程最开始的环节。项目的目标定义了项目的“原因”。基于这一点，你才能定义“什么”是项目的范围。一旦你理解了目标和范围，你就可以开始确定实际的设计规范，即“如何评估”。这些方面都在第二部分详细讨论过了。

9.2.2 定义工作过程

虽然表面上数据库系统涉及的主要是数据的存储和访问，但它们大多数都支持一个或者多个工作过程。用户不会仅仅为了存储数据而存储数据，他们实际想要以某种方式来使用它。理解数据需要支持的工作过程是理解数据模型语义的关键。工作过程将在第12章中讨论。

9.2.3 构建概念数据模型

比起简单的表结构集合，概念数据模型为整个系统定义了数据的用法。其中不仅包括数据的逻辑模型，还包括工作过程与数据交互的方式。概念数据模型将在第13章讨论。

9.2.4 准备数据库模式

数据库模式将概念数据模型转变为物理形式。它包括系统中需要实现的表的描述，以及数据的物理架构。物理架构和数据库模式将在第14章详细讨论。

9.2.5 设计用户界面

无论你的系统技术实施得如何优秀，如果用户界面十分笨拙、复杂或者不友好，那么该项目是不可能成功的。毕竟对于大多数用户来说，界面就是系统。用户界面设计将在第四部分讨论。

9.3 关于设计方法和标准的提示

我并不热衷于为计算机系统的设计制定清单或者按部就班的程序。以我的经验来看，它们事实上会妨碍好的设计，因为分析人员很可能仅局限于对选项框打勾，而不是理解用户的需求。

但是，在大型系统中会有多名分析者以及多个开发团队，这就有必要建立一些管理过程的通用程序。已经有一些现成的方法，并且大多数还有自动化的工具支持它们。我不打算作任何推荐。首先，这会导致信仰的问题；其次，虽然命名不同，但方法的存在通常要比方法的选择要重要得多。

不过我的确理解准备设计文档是一件十分头疼的事情，特别是你刚开始做的那几次。第14章包含一些关于这个过程的一般性讨论。

第10章 定义系统参数

“设计一个系统是为了完成某些任务，而不是所有任务。”

——Robert Hall，研究工程师，北美航空系统

这个故事说的是当一位项目副主管试图第无数次地扩展项目的范围时，Bob Hall对他说了以上这番话，并且这几乎使得Mr.Hall失去他的工作。（在系统设计中，同样在其他过程中，了解你的客户是值得的。）然而，这是我听说过的有关系统设计过程的最真诚的评述。如果项目想要成功，就必须描述项目的目标并合理地划定项目的界线。如果你不能很确定地说明，“我们要做这个；我们不打算做那个”，我敢肯定你将遇到大麻烦。

定义这个过程需要三个步骤：

1. 确定目标，不仅仅是系统的，也是整个项目的目标。
2. 建立系统的设计规则，这将用来判断设计和实现过程中的任何折中方案，同时也用来评定系统的成败。
3. 定义系统的范围——你想要做的以及不想要做的。

10.1 定义系统目标

定义（或者发现）一个系统的目标和范围应当是一个简单的过程。有时候，如果你确实很幸运的话，那么它真的是相当简单。不过有时候，定义项目的目标和范围只是你开始的主要工作内容。更多时候，定义它们是一项复杂的过程，包括正式的分析技术，设计折中方案，以及一些交际活动等。

项目开发的目标通常是决定系统范围和设计规则的重要因素，系统可以依据它来做评价。毕竟，该目标是系统实现的原因所在。很显然，在明确了你将要达成的目标之前，你不可能依据项目的其他方面来做任何正式决定。

不要把“目标”和“总体概述”混淆了。大多数项目开始都会有一个关于系统开发的描述以及预算。你的概述是“使得当前的订购系统自动化”，并且你有一年的时间和一百万的资金来开发该项目。但是“使得当前的订购系统自动化”只是一个总体概述，不是一个目标。目标是该项目实施的一个原因，或者是一组原因。

事实上，探讨确定系统的“目标”在某种程度上是一种误导。绝大多数的系统有很多目标，包括一些切实的和不太切实的，并且发掘它们可能需要一些侦查工作。为什么订购系统要自动化？减少成本？让用户加深对公司的印象？使得管理更有效？系统的目标很可能包括所有这些理由以及其他很多理由。

现在，我并不建议你开始探听人们的隐私或者获取公司的机密信息，并且这些目标基本都是“不关你的事”。你不需要知道部门经理由于恐惧而加入了互联网的潮流。你只需要知道为了自我供养，该系统必须将处理一次订购的时间从10分钟缩减到2分钟。

你可能还需要了解那些销售和市场人士所使用的模糊术语，比如“产品定位”以及“控制用户的期望值”。幸运的是，这不需要回到学校学习；你只需要简单地询问客户就可以了。

每个人对这些表述都有不同的含义，所以你不能不问清楚。

不切实际的目标通常很难转变为可以衡量的设计规则。有时候，一点明智的发掘就能使一个不切实际的目标转变成切实可行的。例如，目标“辅助控制用户的期望值”通常表明的是一个客户服务问题，它可以很简单地转变为可衡量的标准——“我们需要告知客户多久可以完成订购”——或者就放弃它。

如果你的客户发现交货时间存在问题，并且他有理由相信这是由于销售人员为了卖出产品而承诺的不可能的交货日期，那么订购系统和管理用户期望值的目标上就有直接的意义。例如，你可以在订购日期和承诺的交货日期之间强制设定一个最小时间约束。但是，如果该问题是出在制造过程的质量控制方面，那么订购系统则不可能起到任何作用，并且你有义务对你的用户明确指出这一点。这并不意味着该项目不值得进行，但是，每个有关的人员都应明白该订购系统不可能直接影响该预期的目标。

当然，并不是所有的不切实际的目标都能很容易地转换。“定位”就是其中之一。你可能被要求建立一个网站来“定位公司处于技术前沿”。大多数这种类型的目标要么是没有任何实际意义，要么一个系统存在的事实就足以满足它。很容易确定是否是这种情况：只需要询问你的客户你如何得知是否达到了目标。如果其他有关性能和功能的目标达到的同时，该不切实际的目标也能满足的话，这将是最好的情况。（如果你还能发现一名市场人员在实施骗局，这不也很好嘛。我喜欢这样，你呢？）

另一个需要仔细解释的观点就是一个预期的目标一般都类似于需要“提高”和“降低”什么。“增加效率”和“提高生产率”是十分常见，并且是十分模糊的目标。如何得知你是否达到该目标了呢？我的一个客户曾经告诉我一个十分精彩的（同时几乎也全是杜撰的）关于工作要求中可衡量的重要性的故事。（毕竟，设计规则和工作要求的目的是是一致的。）这个故事说的是，一个年轻的销售人员被告知他工作的一部分就是“改善我们的产品和服务”。因此，他打开办公室的门并且喊道，“每个人都应该买我们的产品因为它十分棒”。我相信这不是他的经理所想要的结果。

在最初分析过程中，除非你认为你的幽默感比你的银行账户余额更有价值，否则你必须确定那些需要提高的程度。增加效率到什么程度？将生产力从多少提高到多少？但是这里存在另一个需要小心的陷阱。目标能够被直接衡量当然是好的，并且“将处理一张发货单的时间从10分钟减少到2分钟”的说法显然比“增加效率”要好得多。但是第一种说法假定你已经知道当前一张发货单所需的处理时间，并且发现这的确是一笔昂贵的花销。

研究的代价经常可以避免犯错误的风险。在我们的发货单例子中，很可能没有必要派出一个分析人员小组用秒表来精确测定处理一张发货单所需的时间，虽然我看到过有这么实行的。很多年前，我参与了一个项目，一个政府部门花费\$ 50 000来决定是否应该购买一个过时的绘图软件，而这个软件的推荐零售价只有\$ 2 500。（我得承认，我曾经很明确的指出了这一情况，而那之后我始终对银行都不够诚实。我认为由一个政府部门出钱做一些愚蠢的事情，等同于一种退税方式。）

将这些一般要求转变为切实的设计规则需要依据对于“足够好”的范围和定义的意义理解。如果你将公司或者某人的职业前途都赌在一个新计算机系统的实现上，那么你最好非常、非常明确你在做什么。如果你在构建一个小系统，而它不会对公司的关键产生主要影响，那么你可以试着更随意些。回到我们的例子，很可能“足够好”意味着当前平均每个人一天

能处理25份发货单。这里就没有必要实施更细致的调查了，因为部门经理已经十分确定地告诉你这些了，这就是你被叫进去的原因。我相信过于昂贵的研究是保卫部门如何一次订购了价值\$400的螺丝刀。（并且，在你给我发送抱怨邮件之前，我实在不知该如何处理它。）

总是有必要询问一下某个改进的原因。比如，公司可能存在处理积压产品的问题，经理对此要么加快处理过程，要么雇佣额外的员工。如果你知道该积压问题以及计划在销售中提升（正如我们的发货单例子），你就可以决定实际需要改进的程度。

你所完成的目标可能和你的客户初始给你的有所不同。显然，如果你的客户想要减少一半的处理时间，那么你不得不竭尽全力达到这个目标。但是如果你知道系统实际上只需要达到25%的缩减量，那么当你想要在处理的统一费用和系统的可靠性和易用性之间作权衡时，你就有进行协商的余地。

你常常会听到计算机行业的人们谈论如果他们的客户知道他们想要的，那么他们的生活将会轻松很多。你的客户当然知道他们想要的，他们只是不知道如何将这些需要转化到计算机系统中。那是你的任务。你的客户并不打算有意提一些不合理的要求来误导你，或者责备你那些并不是你做错的事情。但是你工作的一部分就是帮助你的客户确定提出的这个数据库系统能够帮助他们做什么以及不能做什么。期望他们知道当前技术的功能和限制是不切实际的。

关于这个问题的不同形式是客户提供给你一堆屏幕界面和样板报告，这些可能要或者可能不要在系统中实现。这就是所谓的方案而不是问题。这需要一定的技巧来查看在该“系统设计”背后的合理性，而不是去暗示创建它的人是愚蠢的、不胜任的，或者干脆就是做错了行业。

我所能建议的是这些情况就好比你在检验水一样。如果你的客户好像在拒绝你的问题，那么试试说一些类似这样的话：“如果我对你的业务环境理解得更清楚些，那么我将能更好的帮助你。”如果这个方法不是在任何地方都奏效的话，你就不得不像要求的那样来实现系统，或者离开这个项目（我意识到通常这是不可能的）。你能期望的最好的方法就是重新检查提供给你的设计，并且如果你发现某些根本性的错误，那么以这样的方式来讨论它们：“我做不到这样，但是我可以做成那样或那样。哪一个最切合你们的要求？”

之前描述的抽取目标的过程对数据库系统并不奏效。数据库系统与其他计算机系统最主要的不同之处在于它有一个关于企业的数据主体，并且几乎都是作为副产品。该数据主体不论是一个用户列表还是一组发票，对于企业来说，它都具有内在的价值并高于系统直接支持的工作过程。

当然，我并不是建议将每个项目都看作是一个创建企业级数据知识库。我只是想说作为系统的一个部分，数据应当根据企业的其他领域或者有着相同领域的其他工作过程来检验其价值。这可能使你的系统能够根据其他领域来累计数据，不过老实说，这种机率远比想象的要小很多。

例如，如果出货系统维护一个客户列表，而销售部门需要客户的邮寄地址以便寄送产品的实时信息，那么对他们作一个列表是十分合适的。如果没有其他事宜，那么共享该列表可以使一些可怜的文职人员免于重新录入这些客户的名字和地址。但是请注意“使得数据可用”并不意味着要将该邮寄列表函数整合到发货系统中——这是十分糟糕的。

假定它充满了浮夸的危险，那么我们建议考虑该问题的唯一原因就是数据结构作微小的改变以适应其他的用途，这种方式可能很合适。我们将在第12章详细讨论这种可能性，但

是让我先在这里给你一个简单的例子。

在谈论原子值时，我说过在一个给定系统的语义下，一个地址可能就是印在一个邮件标签上的斑迹。我建议在这种情况下，你考虑将该地址看作一个单一的属性。但是，如果仅当将该数据分开成几个属性时，它就会对其他的领域也很有用，那么为了避免在其他地方重复录入数据，在当前系统上多花一点额外功夫是合理的。

但是请注意，额外的功夫应该是很少的，并且共享这些数据是可行的。我曾看到过（并且让我觉得羞愧的是，它甚至实现了。）这样的系统，它要求输入整个类别信息，而这些信息与手头上的工作并没有直接联系，这么做的原因仅仅是由于它们会在某个时间对某个人有用。这是非常容易发生的，所以一定要确保在你谈论“为了将来的增长计划”时，你的实际含义不是指“增加不必要的负担”。

一旦你已经建立了最初的一套目标，那么你就可以进行下一个分析过程的工作：确立设计规则和项目范围。但是，不要错误地认为该目标是一成不变的。你必须时刻准备在之后的项目过程中重新评估你的系统目标。

对于任何超过若干星期的项目，业务需求是一定会变的。销售可能从开始就会有很大变化；公司合并可能意味着人员过剩而不是不足；任何外部事件都可能要求重新评估项目的目标。现在就和你的客户检查并且在一个长期的项目过程中，要确保不会有十分根本上的变化。

甚至在一些相对短期的项目中，你可能发现在项目的后期，一些目标或者不合适或者无法达到。要记住，经典的瀑布模型中的一个主要问题是它假定在你需要的时候，你可以得知任何你想要知道的信息。在现实中，你将会通过整个项目才能扩展你对系统的理解。新的理解经常需要重新评估系统的目标，即使这只意味着回顾一下目标并说一句：“好的，这些目标仍旧有效。”

10.2 开发设计标准

在你已经对系统的切实和不切实的目标有了合理的理解之后，你就可以开始开发设计标准了。当然，事情总不会这么十分顺利有序，并且在实际中，你会在定义系统目标的过程中积累设计标准。但是出于探讨的原因，我们假定你已经确定了一系列的项目目标，并且现在你需要准备一个关于标准的列表，系统的成败要依据它来进行评判。

设计标准和项目目标十分贴近。如果该项目的目标告诉你该向哪个方向进行，那么相应的设计标准就是告知你是否达到了该目标。一个系统的所有设计标准都应当直接支持一个或多个系统目标。如果你发现一个重要的标准与系统目标不匹配，那么这必然意味着你的目标列表不够完整。

并不是严格要求每一个标准都要匹配它所支持的目标，但是这是一个十分有用的做法，甚至对于那些有经验的分析人员来说，这也是很有益的。任何项目的最严重的问题之一就是你不清楚你不知道哪些内容。这就好比你是一位对该企业的活动没有太多（或任何）经验的外聘顾问一样。目标和标准的不匹配可以很好的说明你还没有理解你需要理解的所有内容。

设计标准一般都采用以下三种形式之一：

- 直接衡量需求，比如“在不超过两小时内打印一个完整的报表”。
- 环境标准，比如“在现有的LAN上进行操作”。
- 一般设计策略，比如“提供上下文相关的用户支持”。

这些具体的分类并不十分重要，但是列出的这些对于判断你对系统的理解程度是很有用的指示。大多数的设计标准都应该遵守第一个和第二个类别。如果你只有一个设计策略的列表，我敢打赌你对将要解决的问题并没有一个足够清晰的认识。

注意：不论采用哪一种形式的标准，一旦你已经符合它们了，就立即停止。你的项目已经完成了，应该举行个派对庆贺一下。这个命令并不像它听起来那么轻松。举个普通的例子，你正在优化一段特殊的代码。为了符合设计标准，该函数必须在10秒钟内计算出一个特定的值。你已经将时间减少到了9秒，但是你确信如果你采用其他方法可以将时间再减小一半。不要这么做。或者，如果你非要说的话，就利用你私人的时间进行。你必须在所有项目标准都达到之后才可以进行，否则该项目永远也无法完成。

实行这种规则的唯一可能的例外情况就是研究和开发（R&D）项目，但是R&D项目也有不同的目标，因此也就有不同的标准。与“在10秒钟内完成该计算”所不同的是，一个R&D项目的设计标准更像是“确定最佳的计算方法”。由于你可能永远不能确定任何给定的方案是否是最优的，所以你可能永远也无法满足这个标准；因此你可能永久地探索下去或者直到你耗费完所有资金，不论哪一个先发生。

重要的是，在你开发设计标准的时候，不能突然就确定某些特定的设计或者架构。你可能十分确定你将要使用Microsoft Transaction Server来支持系统的可伸缩性，但是这是一个架构决定，而不是一个设计标准。设计标准应当是该系统需要可伸缩地支持x个用户。

当你尚在犹豫中时，请记住一个设计标准是用来决定一个项目是否成功完成的。在这个例子中，问问你自己：“如果我们使用Microsoft Transaction Server，我们的系统就完成了吗？”可能会完成，但是事实上，你使用Microsoft Transaction Server并不能告知你这个问题的答案。但是，问题——“如果我们能够支持x个用户，我们完成了吗？”——的答案就可以，当然，前提是你已经满足了其他设计标准。

10.2.1 直接衡量标准

我已经讨论过确定客观衡量目标的重要性。如果你在这方面做得很成功的话，那么你的大多数设计标准必然也如此。如果一个目标是减少50%的处理时间并且当前处理时间是10分钟，那么该设计标准显然就是“使得处理在5分钟内完成”。

有时候，很难将衡量目标和直接衡量标准区分开来。我并不认为这个问题十分严重。规范部门不会因为你列出的内容既是一个目标又是一个标准而将你拘捕。当然，如果你没能使得一个或多个标准支持衡量目标，他们可能会给你一个警告。

注意不要对你的衡量标准作微管理。可能有这样的情况，一个特定的处理需要1分钟完成，一个给定的查询必须在10秒钟内执行。但是那是一个实现问题，而你对项目的了解程度还不足以让你能做有关实现方面的决定。

10.2.2 环境标准

大多数的环境约束是描述已经存有的操作环境和任何你必须与之交互的合法系统。不太可能在一个空白的环境中详细说明一个系统。大多情况下，你的客户会有一个期望系统所运行的硬件和软件环境。

环境标准的其他主要来源是数据库系统特有的：处理的数据量。早期我作为一名独立顾问时，有过一件不光彩的事情。我在为一个计算机硬件批发商在某个区域的分公司的销售跟踪系统准备一份引证。在讨论过他们的要求之后，我提交给他们的论证说该系统是只需用 Microsoft Access 2.0 就可以实现的小系统，而我仅被告知他们想要跟踪整个公司的销售情况，它有 500 个地区分公司和数以千万美元的销售额——显然这远远超过了 Access 的处理能力。我原来以为他们只需要跟踪那一个分公司的销售。（不用说，我没能获得那个工作。）

对于数据量，你需要检查两个问题。一个是绝对的数据量，另一个是它的增长方式。一个图书馆可能有百万的数据量，但是每天只会增加很少的记录。一个订购系统可能每天只增加几百条记录，但是在销售完成后都要将这些记录存档。显然，这两种方式需要不同的设计策略。

证明过度设计系统的一种情况就是所支持的数据量。作为一种一般规则，我建议计划的容量至少比客户提出的最大值大出 10%，并按四舍五入计算。对于稍小的系统，我认为应当增加 20% 至 25% 的额外容量。

数据量在有更大数据容量的情况下就不再是问题了。一个设计良好的客户/服务器系统支持 100 000 条记录和 10 000 记录一样简单。但是一个基于 LAN 的 Access 系统使用 Jet，它最初被设计成用来支持数千条记录的数据量，因此不论它设计得有多好，也不太可能扩展到支持数百万的数据。

另一个环境标准的主要来源是系统需要支持的用户数。大多数系统都有超过一种类别的用户，并且需要为每一种类型的用户定义需求。例如，订购处理系统显然要有用户来输入订单的功能。它很可能还有一组用户需要查询订单的状态并且可能更新数据，而第三组用户需要从整个数据库中生成报表。这里的每一类用户都需要系统不同的支持，因此每一类都需要在标准中分别说明。

你还必须区分两种用户，一种是连接到系统的用户，另一种是实际使用系统的用户。例如，Jet 数据库引擎限制在任一时间内最多只能有 255 个用户连接到数据库。这意味着最多允许 255 人同时打开数据库。它并不是说 255 人可以同时更新数据库。

10.2.3 一般设计策略

一些项目目标不能很容易地转换为简单的数值衡量标准。例如，一个类似“提高数据输入精度”的目标就很难量化，因为在这种情况下，确定发生了多少错误所耗费的成本很可能超过带着一定数目的错误来衡量你的成功而获得的利益。

你不应该忽略这些类型的目标，你可以将它们以设计策略的形式描述出来而不是衡量标准。在这种情况下，设计标准可能是“通过允许用户在可行的时候从列表中选择来提高数据输入的精度”，或者是“通过在接受发货清单前实现恰当的信用检查来减小信用异议的概率”。

和可衡量的设计标准一样，你不应当在这里太具体并突然作一些实现方面的决定。你不是在设计系统，只是在建立一些标准，依据它们来评价系统的成败。上述的例子谈论的是作一些“在可行的时候”的事情和实施“恰当的检查”；更具体的事宜需要在更好地理解系统需求后的设计后期阶段来定义。

但是你也应该避免一些空洞的表达。“该系统必须是用户友好的”，这听起来很不错——毕竟，没有人愿意面对一个敌对用户的系统。但是这不是一个有用的标准。确定一个系统是否

符合标准——“该系统将遵循《软件设计的微软界面指南》”——则是可能的。但是，系统是否用户界面友好是一个经常争论的话题。你当然想要减少而不是增加与设计标准之间的差异。

10.3 定义系统范围

一旦你明确了实施一个系统的原因，你就有依据来决定哪些功能可以合理地落在系统范围内（以及哪些不在系统范围内）。如同设计标准一样，系统范围内的功能应当直接支持一个特定的目标。

例如，提高销售订购处理的效率是一个项目目标，打印发票直接支持该目标，很明显它是属于预定系统的范围。而另一方面，生成一个产品目录则不是，因此它不在系统范围内。即便当该目录是由同一个人生成并且与订购处理系统共享一定量的数据时，它依旧也不属于系统范围内。

有时候，当核心和“外来”目标之间的数据或者功能存在很多数据重叠时，就意味着需要重新定义系统目标了。在前面的例子中，如果该系统变成是一个“销售支持”系统而不是一个“订购处理”系统，那么“产品目录”将成为项目目标之一。

但是这也同时是一个危险的过程。你不要期望使用预想的系统范围来定义目标。这就好比在马前面放置大车一样。从经验来看，通过扩展系统的范围来包含一些简单或者有趣的功能是十分诱人的。同样由经验可知，当用户询问“为什么我要这么做？”，而你不能给出合适的回答的时候，你会有多么尴尬。

Norton Utilities的旧版本——在MS-DOS时代不可或缺的一个软件——就是一个典型的例子。该程序过去通常在退出的时候报告它已经运行的时间。通过好几年对这个功能的思考，我发现它这么做的唯一的原因就是“他们可以做到”。

我认为这么做是不合适的。规则是如果一个功能不能直接支持一个目标，那么它就应该在系统范围之外。和其他大多数规则一样，这条规则也可能被打破，但是你应该只有在仔细考虑之后再这么做，并且你确信该功能确实有一个合理的用法。对一个软件的装载进行时间统计可能是一个很酷的功能，但它几乎没有任何用处。

还存在这样的情况，该功能对用户来说有明显的价值，但是这个价值超出了实现它所花的代价。之前描述的产品目录的目标可能就是关于这方面的一个很好的例子。

但是即使是在如此鲜明的情况下，你也很可能要扩展系统的目标，不过你最好将这些额外的功能包含在一个通用范围类型中——“附件的有价值的功能”，这么做会更安全些。这就很清楚的表示该功能并不是绝对必要的，并且如果日后你发现它的实现成本将比预期的超出太多，或者你已经没有时间和资金了，那么这些功能可以很容易地从系统中去掉。

第二种你可能需要包括那些不直接支持系统目标的功能的情况是当客户坚持要求包括它的时候。可能看上去，对你来说生成一个雇员电话列表与处理订单之间显然没有任何联系，但是如果客户想要，那么他们就需要它。你可以指出该功能并不支持任何具体的目标，但是你的工作就是尽力满足你的客户的需求，而不是定义它们。

成本-利润分析

在设计过程这个阶段的最后一步，你会发现对范围内的功能实施一个成本-利润分析是十分值得的。特别是当预定的系统有多个组件时，这会更加有用。不同组件相对的成本-利润比

例可以帮助你确定这些组件实现的顺序。并且如果你考虑扩展系统范围，将“增加的有价值”的功能包括进来，那么成本-利润分析就可以作为系统可行性检查的一种手段。

和其他事物的道理一样，利润/成本比例较高的组件应该首先实现。这种策略是最佳的开始方式，它允许系统尽早地开始自我供给。这对于那些需要花长时间完成的项目特别有效。如果可以迅速交付核心功能，那么就有一个很好的基础来评估未来的开发工作，并且减小了由于业务环境的变化而导致系统之后不可用的风险。

如果系统可以在开发过程的早期就开始自我供给，你可能还会发现投资那些“有趣但并非严格必要的”功能成为可能，而这些功能在定义系统范围的时候是被清除在外的。当然，相反的情形也会偶尔发生。客户可能发现这个最初实现的功能已经足够满足他们的目标，从而无限期地拖延未来组件的开发工作。

当然，实施一个正式的成本-利润分析服从这样一条规则：犯错误消耗的成本超过研究答案所花的成本。成本-利润分析并不困难，但是它们是很耗时间的，显然花两天的时间去分析一个只需要一天就能完成的系统是不够明智的。在很多情况下，一个正式的分析（另外又被称作内部测试）就完全足够了。

虽然原理很简单，但是存在多种实施成本-利润分析的方式。用估计的功能成本除估计的功能利润将会得到一个数字值。这个值越高，该组件与其他组件相比的价值就越高。

注意：在项目的这个阶段，我们谈论的是估计成本和估计利润。但只有在一个功能实现后你才可以确定地得知它的成本，同样，你只有在该系统被使用一段时间后才能确定它带来的利润。后现代主义的分析对于提高未来估计十分有用，但它们是一种完全不同的活动。

在决定通用的衡量单位时，成本-利润分析会遇到一些麻烦。所有的成本都必须用相同的单位来衡量，并且所有的利润也必须使用相同的单位衡量，虽然这两个单位没有必要一样。例如，你可以用成本的时间值来与货币值作比较。该结果比率可以用来与其他使用相同单位计算所得的值进行对比，这样的结果是完全有效的。

估计成本通常可以使用时间或者货币值来衡量，并且这二者在一个商业环境中通常是可以相互转换的。因为这两者在某种程度上都是现成的，但是，最好是以某些派生的值来描述成本，比如“工作量”。这就避免了将一个成本-利润分析和一个报价单或者一个实施进度表相混淆的可能。

对利润使用通用的衡量单位可能会产生更多的问题。例如，你可能估计自动化一个给定工作过程将提高20%的效率，并且减少50%的错误。这个20%的效率提升可以转变成存储的时间值，并且如果有必要的话，可以由此转变成货币值。但是给提升精度指定一个值就不太容易了。也许可以估计查找和修订错误的成本（不论是以时间为单位还是以货币为单位），但是对于其他难以确定但又是非常现实的利润问题，这种做法就不太可能了，比如估计修正一个客户订单所带来的利润。

在这些情况下，你可以使用多种方式估计利润。例如，你可能以“储备的资金”、“盈利的金额”以及“无形的利润”来估计利润。然后，你可以计算三个比例，每个值一个。这可能会使得比较有一点复杂——你和客户可能会发现很难确定该值为3/6/2的一项功能是否比另一个值为6/3/2的功能要优越。在这种情形下，你可以以某种方式将这类值标准化，以此派生出一个单独的成本-利润评估。

如果它们相对来说都很重要，你可能会决定为每一类利润增加一些数值。不可否认，如果你添加的是苹果和橘子，那么在这个意义上来说，将它们简单地认为是水果是可以接受的。但有时候，一个平均值会更合适些。

正如经常发生的一样，如果这些类别对于企业的重要性是不同的，那么你可以为每一类别指定一个相对值，并且将每一类别的利润与该重要性因子相乘。例如，就之前引用的例子来说，你可以决定“储备的资金”并不十分重要，但是“无形的利润”比较重要，而“盈利的金额”是“无形的利润”的重要性的两倍。因此，你就可以指定修正因子1给“储备的资金”，2给“无形的利润”，以及4给“盈利的金额”。其结果如表10-1所示。

表10-1 一个带权值的利润分析样例

	盈利的金额 (修正因子:4)	储备的资金 (修正因子:1)	无形的利润 (修正因子:2)	总量	盈利总量	平均值	平均盈利
Function1	3	6	2	11	22	3.6	7.3
Function2	6	2	3	11	32	3.6	10.6

一般以相对值而不是绝对值的形式指定利润估计会更容易些。例如，通常很难给功能X无形的利润指定值3。但是可以说功能X的无形的利润值是功能Y的两倍，而功能Y和功能Z有相同的无形的利润值。

成本-利润分析对于获取系统的预定利润是十分有用的工具。它们提供了一种简单的方法来比较不同组件之间的相对价值。但是它们仅仅是工具，只是基于最佳猜测的估计值，因此绝对不要误以为它们是绝对值。即便功能X的利润比例为12，而功能Y的利润比例为2，也仍旧有可能需要（或者技术上需要）首先实现功能Y。

一个成本-利润分析的结果必须结合其他的因素来查看，比如系统的依赖性。它们还需要以你对系统改进的理解来检查。一定要在开始每一个组件的工作之前重新评估你的估计值，你可能会发现对以前组件的实际成本和利润的相关经验会改变你对未来组件的估计。

10.4 小结

在这一章，我们探讨了在项目开始前涉及理解系统的有关活动。你必须首先确定系统目标，然后将这些目标转变成设计标准，它们被用来评价项目的成败。你还必须确定系统的范围，它是确定需要或不需要作为项目的一部分实施的界限。

这些活动好比是步骤0，它们是那些在开始设计系统前你必须要认真做的事情。在下一章，我们将详细讨论设计过程的第一步：定义系统将要支持的工作过程。

第11章 定义工作过程

虽然很多数据库系统都被设计用来简单地存储和检索一些数据集合，但绝大多数的设计都是为了协助执行一项或多项活动。这些活动就是系统将要支持的工作过程。一个工作过程简单来说就是一项或多项离散的任务，它们合起来描述对于企业来说十分有意义的一些活动。“处理一张订单”和“查询一位客户的电话号码”都是工作过程的例子，虽然它们的复杂程度不一样。

一项任务是一个离散的动作，是工作过程实施的一个步骤。例如，销售订单的处理过程可能由多个任务组成：“记录销售订单”、“检查客户信用”、“检查库存情况”以及“发送订单”。然而，查询一个客户的电话号码的过程可能只有一个任务：“查找客户记录”，或者，如果你想要更详细一些，则是“查找客户记录”以及“显示客户记录”。

区分一项任务和一个过程有时候是很困难的，它们的差别相当随意。就像在数据模型中确定哪些描述了一个标量值一样——在一个模型中的一个单一属性可能在另一个模型中被分解为多个属性。在一个系统中的一个活动可能在另一个系统中被认为是一个过程，并且在低一级的细节程度上被分解为多项任务。如同数据建模的过程一样，这个决定必须建立在对问题域的语义理解上。

一些系统不适合工作过程分析，特别是报表工具，它们不能像支持某种类型的活动那样支持一个特定的过程。在这种情况下，构建用户情景会更加合适。这些会在本章的最后讨论。

11.1 确定当前工作过程

在分析业务过程中，第一步当然是定义系统范围，因为该定义会告知你需要分析的过程。检查系统范围内不同过程的顺序通常是不重要的。即使你计划某些组件的实现在其他组件之前（增量开发），至少也应当在你开始实现之前对所有工作过程实施一个粗略的分析。这么做可以使你发现过程之间的依赖性，这会影响到组件实现的顺序。

11.1.1 与用户交流

在确定系统范围内的工作过程之后，下一项任务就是获取当前所有已经完成的事宜，以便执行它们。在这一点上你不需要太担心哪些代表一项任务，哪些代表一个需要进一步分析的工作过程。只要找到一些可以告知你的人就行，“好了，我们从销售人员那里得到这份文档，首先我们需要浏览它以确定他们是否正确地完成了它；如果他们完成了，那么我们找到客户文档，然后……”，尽可能地问一些问题，然后记下它们。你还应该复印早期在过程中用作输入或者输出的任何形式的报表。你的目的是理解发生了什么，而不是分析该过程。

顺便提一下，很多人称这一分析阶段为“用户面谈”。我更倾向于“过程分析”或者其他中性的术语。人们总是很容易低估计算机带来的威胁，即便是对那些正在使用它的人们。很多人仍旧在担心他们将会被计算机替代，因此“用户面谈”很容易被误解为，“我们将决定谁能获得工作职位”。这种情况在大企业中格外明显，在大企业里，不太可能和每个人对话，并

且很多人可能都不确切的知道你是谁或者你打算干什么。

任何可能的时候（并不是所有时候），你应该尝试与那些真正实施该过程的人员交流，而不是和经理或者主管交流。我的经验是经理们都倾向于在他们的权限下谈论对工作过程的理想观点，而那些每天都进行一项工作的人是最佳人选，他们能告诉你他们遇到的问题和阻碍。当然，你也应当与主管们交流，因为通常他们对实施具体任务的原因，以及不实施它们的原因或者错误地实施它们所带来的后果有最好的理解。

在你们的讨论中，务必要发现那些对该过程可能的例外。例如，如果用户说，“我们对订单的完整性进行检查”，那么就要确定如果它不完整将会发生什么样的事情。有可能他们只是跳过它，但是你需要知道他们是否需要自己去发现信息，也许你的系统能够使这个过程更简单一些。事实上，对于用户实施的每一个活动，你都应该询问可能会发生什么错误以及错误发生后要如何处理。

既然我们在这里特别关注数据库系统，那么还应该特别注意数据在过程中是如何使用的？信息的使用量是多少？它们从何处获得？它们是何种形式？如果它们没有表达出来或者以错误的形式表达将会发生什么？这些问题的答案就形成了概念数据模型的原始资料，我们将在下一章讨论这一问题。

很多组成工作过程的任务都是由不同的人员实施的。显然，可能的话你应当与每一位涉及其中的人员进行交流。这个建议也适用于那些任务显然在系统范围之外的人们。比如，拿之前描述的销售订购过程的例子来说。“发送订单”任务可能事实上就是“将订单发送到运送部门”，因此至于该订单在运送部门会发生什么可能就是系统之外的事情了。但是，与运送的人员交流一下也是一个很好的主意，这样可以确定他们得到了他们需要的所有信息并且这些信息呈现的形式对他们是有用的。

类似的，如果在过程中需要不时地打印某个报表，那么你需要找那些拿到报表的人员并了解他们将如何处理这个报表。你会十分惊讶在企业里会有很多纸张没有理由的在传阅。（当然，也许你不会。）或者，更经常的情况是，虽然应当存在该报表，但是它包含的却是错误信息，或者它包含正确的信息，但其形式却是错误的。因此它可能在其他地方被整理归档了，但并没有像它所设计的那样付诸使用。

11.1.2 确定任务

当你和相关人员——他们当前正在实施的工作就是你的系统将要支持的——交流过之后，你应当对所涉及的活动有了一个合理的理解。下一步就是将这些信息组织成一系列的任务，其关键在于确定应用于该过程的业务规则。

在本章的开始我们将术语“任务”定义为一个离散的动作。现在在本节可以进一步精确地定义“离散”的含义。它有两重意思：一是该动作有明确的开始和结束；二是有关的业务规则在任务开始之前和任务完成之后都是有效的。然而，这些规则可能在执行任务过程中会被临时打破。

一个业务规则就是一个起源于问题域的约束，它与那些派生的约束相对应，例如一个数据类型。因此，“订单的运送日期不可能是4月36日”不是一个业务规则，因为它依赖的是日期域。（不管怎么说，这的确是一个很傻的规则。）但是，“订单的运送日期不能在订购日期之前”依赖的就是问题域；它是企业业务方式上的一项功能。因此它是，或者至少可能是一项

有效的业务规则。顺便提及一下，即使该企业不是做业务的，术语“业务规则”也同样适用。哪怕你构建的数据库是用来跟踪你收集的古董顶针，它仍旧服从业务规则。

大多数的业务规则与数据处理的方式相关。“客户的邮政编码不可以为空”，“发货日期不得早于运送日期”，这些都是数据相关的业务规则的例子。其他的规则，例如“当订单超过客户的信用额度时，需要销售经理的授权”，就不直接约束一个数据值，虽然它们可能由一个数据值触发，就像这个例子一样。

不用担心，发现属于一个工作过程的业务规则并不像想象的那么困难。这类似于“它如何会出错，以及如果出错如何处理？”之类的问题包含的内容。你不需要在这一点上过分担心规则的细节。细节是构建概念模型的一部分，这是之后需要做的。在这里你所要做的所有事情就是将你确定的不同的活动分组，这种方式使你可以合理的确信这些业务规则不论在该动作的哪一方面都会是有效的。

让我们来看一个例子。比如你列出的处理一张销售订单的动作包含以下任务：

1. 检查销售订单的完整性。
2. 如果该客户已存在，检索客户资料。
3. 记录运送信息。
4. 输入订单细节。
5. 为新客户指定客户代码。
6. 检查订单项目的可用性。
7. 检查客户信用额度。
8. 挑选订购货物。
9. 包装订单项。
10. 准备运送的文档。

首先需要注意的是该列表中的事项似乎是以某种随机顺序完成的。这没有关系，你只要尝试理解当前需要完成的任务就可以了，之后再整理该过程。此外，其中的一些项目似乎包含不同的细节程度。我们之后马上讨论这一点，至于现在，我们仅仅要确定任务。

项目1，“检查销售订单的完整性”，有一个离散的开始和结束点。该动作在接到新订单的时候发生，并且所有文档检查完毕之后完成。我们可以假定在一个过程开始的时候所有的业务规则都是有效的，因此这里就不存在问题。如果初始的销售订单文档不符合业务规则，它将会被拒绝。因此我们知道如果该过程继续到下一阶段，该规则一定是有效的。所以项目1是一个任务。

属于项目2“检索客户资料”的业务规则只有一项，就是查询资料的个人必须有访问它的权限。我们假定实施该过程的所有人员都有访问资料的权限，因此这个任务准则不适用。该动作在销售订单被检验后发生，但是由于这些资料在之后的动作中还要使用，所以它没有一个离散的结束点。因此它不是一项任务，它只是一项任务中的一个步骤。

事实上，该用户资料会在项目3（“记录运送信息”）和项目5（“为新客户指定客户代码”）中用到，这些事项都是属于同一任务的第一个线索。实际上，很明显项目2到项目5可以综合为一个单一的任务，可以称作“记录订单”。在这个例子中，项目4（“输入订单细节”）显然是记录订单的一部分。但是，请不要惊讶于一些任务可以同时实施。在一个手工系统中，某人同时填写两张表格是很容易的事情。事实上，这些表格逻辑上属于不同的任务是可行的。

现在我们有一项新任务，它由四个离散的步骤组成。该任务开始于检查原始文档的完整性，结束于整个订购记录完毕后。但是这里有一个问题。你的客户不允许消费者的订购超过他们的信用额度，但是信用额度直到项目7才能检查，“检查客户的信用额度”是在检查订单项目的可用性之后。然而，直到该订单确定在客户的信用额度之内，你才能确信这些业务规则在任务的任一方面都是有效的。因此项目7是“记录订单”任务的一部分。

不过，项目6，“检查订单项目的可用性”，逻辑上不是记录订单的一部分。事实上，项目6描述了一个离散的任务，它开始于“记录订单”任务完成后，结束于确认有足够的库存来运送。因此项目6自身就是一项任务。不要说我没有警告过你。

在类似这种情况中，一定要确定这些任务看起来无序完成的原因。大多数只是为了便利，但是偶尔这些不一致性会导致一些操作上的约束，它们会对你的系统处理过程有一定的影响。

在这个例子中，如果项目的可用性检查仅仅只因为逻辑上实现起来很简单就被提前了，那么你可能要重组这些活动。但是你可能会发现在订单录入过程和生成过程之间存在一些交互，它要求立刻检查库存级别，并且这种交互必须要在你的系统中得到调整。

如果我们迟一些才发现由于库存不足致使订单不能接受的，那么“检查订单项目的可用性”的任务将变成“记录订单”任务中的一步。正如我们将在下一节看到的，在任务和逻辑级别之间转移活动是相当简单的过程。这里的关键问题是确定这些活动。

分析接下来的三个项目——8、9和10——依赖于系统的范围。如果运送产品是系统开发的一部分，那么就er必须仔细地察看这些事项。通常情况下，如果系统范围止于完成订单并交付运送部门，那么这些项目仅需要简单地混作一个单一的任务：“提交订单”。

当然，我们可能知道在这之后还有若干项目，并且也没有理由将这些信息弃之不顾。事实上，列表中的最后一项，“准备运送文档”，可能甚至是一个完整的工作过程。但是所有这些任务都不属于该项目的范围，因此我们列出它们只作为我们保存的所知的信息，但是我们不会考虑它们。我们修订之后的任务列表如下所示：

任务1：检查销售订单的完整性

例外：如果订单不完整，则拒绝之

任务2：记录订单

步骤1：检索客户资料

例外：如果资料已不可用，则保留旧的

步骤2：记录运送信息

步骤3：输入订单细节

步骤4：指定客户代码

步骤5：检查客户信用额度

例外：咨询销售经理

任务3：检查项目的可用性

例外：咨询销售经理

任务4：提交订单给运送部门

步骤1：挑选订单

步骤2：包装订单项

步骤3：准备运送文档

在将这些活动组织成到各项任务 and 过程中，你几乎一定会发现有些事情并不像你想的那样理解得很清楚，在这种情况下，你应当回到用户那里弄清楚它。无论如何，你都需要和你的用户一起复查所有过程。通常，将这些过程写下来会提示他们提供额外的信息——一些他们遗漏的步骤或者你忘记询问的例外情况。

11.2 分析工作过程

现在如果你对如何完成当前的工作有了一个清晰的认识，那么最好是检查一下所有的工作过程，看是否有提高的可能。正如我之前所说的，大多数企业都有很多的惰性，这一点在文档工作中和在工作过程中同样适用。开发一个新的计算机系统为清理那些无用的东西提供了一个绝好的机会。

有一个旧的效率规则，是说你决不要处理同一个文档超过一次。同很多规则一样，这个规则并不是总实用，但是经常发现一个工作过程可以通过记录任务而简化，因此很多工作就不用在人员或过程之间互相传递了。由于某些原因，类似“我完成A并且将它交给你，然后你完成B之后交还给我，之后我再完成C”这样的次序在实际的工作过程中很难看到，但是当这些活动写下来之后，它们立刻就变得十分清晰了。

为了做这类分析，你必须十分清楚任务之间的依赖性。给定任何过程，一些任务依赖于其他任务，因此必须以某种特定的次序完成。例如，你不可能在你记录订单之前将订单交给运送部门。其他任务的次序可能较为独立一些。例如，你在记录运送细节之前或之后指定客户代码并没有太大关系。

关注数据的依赖关系是十分重要的。一些任务是负责生成信息的，比如客户代码，它就会被用在其他任务中。例如我的一位客户使用类似在前面章节描述的销售订单处理系统，不同的是会计部门建立客户代码并初始化信用额度，而不是销售部门来做，销售部门只负责订单处理。

该客户初始化的工作过程如下：

任务1：检查销售订单的完整性

任务2：记录订单

步骤1：查询客户资料

步骤2：记录运送信息

步骤3：输入订单细节

任务3：实施客户信用检查

步骤1：检察客户的证明材料

步骤2：检察客户信用额度

步骤3：指定客户代码

任务4：完成订单

步骤1：检察订单项目的可用性

步骤2：将订单提交给运送部门

任务3是由会计部门来实施的，只有信用检查的结果合格了，订单才能返回给销售部门。当然，这里存在的问题就是已经完成了对初始数据的录入。这样做不仅意味着如果该客户未被批准，这项工作就白费了，而且还要增加额外的工作，周期性地清除无用的订单。同时，

还会导致会计部门的数据录入人员和销售人员之间的争吵，因为销售人员想要他们的订单都被录入（因为他们可以拿到薪水）。改变任务的顺序，即在订单之前实施信用检查可以消除数据录入人员的低效率和不必要的争吵。

除了确认任务之间的依赖关系之外，还应当检查是否存在不需要实施的任务。这种情况很少，但是有时候如果在各过程中跟踪数据，你会发现有些项目是为其他任务或过程的使用创建的，但是现在不再需要它们了。不太可能整个过程或者任务中的步骤都是多余的。大多数人都很聪明地避免繁重的工作，但是这种情形可能被过程之间的交互掩盖了。生成冗余的报表就是最常见的例子。

当然，这并不是一个大规模改变你的客户商业惯例的借口。如果你的Gertrude婶婶要你给她的针织图案实行计算计划，我也不建议你告诉她应该改变她编织的方式。并且事实上，极少能发现工作过程中明显的无效任务。但是你的工作就是尽你所能帮助你的客户更好的完成工作，而审查工作过程就是其中的一部分。

11.3 将工作过程文档化

和设计过程的其他方面一样，你花在分析工作过程和规范你的文档上的时间与系统的复杂程度是成比例的。一个简单的记录名字和电话号码的系统可能只需要不超过一小时的谈话以及一些你自己使用的速记笔记。

但是，除非你既是客户又是设计者，否则我建议即使是在最简单的项目中也至少需要两次会议。第二次会议的目的是检验你对客户需求的理解，并且确认你已经理解的和计划要实施的都是正确的。

比较复杂的项目可能需要众多人员数周的讨论相对复杂和正式的文档。使用本章介绍的结构化的任务和步骤列表对简单的过程实行文档化还算足够，但对于更复杂的过程，我倾向于使用图片。

虽然业内认为，用作文档化数据关系的实体联系（E/R）图是十分标准的，但它在工作过程图中缺乏一致性。这种图形方法与应用特定分析的技术和项目紧密相关，并且更加时尚。

如果你熟悉这些方法中的一种，并且使用起来很顺手，那么就没有必要改变它。这里，其目的是为了理解和传达信息。UML、数据流图和过程属性图都是很有用的工具。特定的画图技术对我来说更像是些获得有关宗教的无聊东西，虽然人们的确经常这么做。

如果缺少正式的技术工具，可以简单地自行创造。你只需要五种符号来分别表示任务、文档、数据项、决策点以及事件，比如任务的开始点和结束点。我在自己的工作使用的这些符号如图11-1所示。

如果只需要少数几个步骤就能完成一项任务，就可以将它们罗列在一个任务栏中。如果需要很多步骤才比较合适的话，我会将每一项任务扩展为一张单独的图并且分别使用任务栏来描述每一个步骤。有时候，为了更清晰，我会使用阴影或者加粗的线条来表明该任务需要由外部组来完成，例如当会计部门为新客户实施信用检查的时候。

数据项符号可以描述一个单一的属性，比如一个客户代码，也可以描述整个实体，比如整个客户。当项目实际上是由任务创建的时候，你可以使用阴影

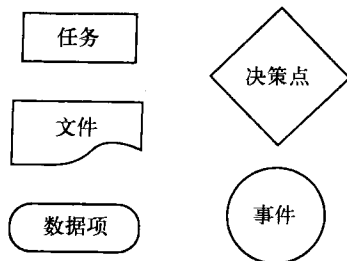


图11-1 工作过程的符号

或者加粗线条指出这一点。一些分析人员还喜欢在数据项被一项任务“消耗”的时候标明它；它的意思是说，该数据项被一任务使用并且在过程中不与其他任务共享。说实话，我极少发现这是有用的信息，并且还是倾向于保持我的图形的整齐有序。

当已经决定你将要使用的符号之后（强烈建议你选择那些能简单地随手画的符号，而不是有什么本质涵义的符号），接下来你需要一种组织它们的方式。我使用箭头来表明依赖关系，使用一个分支线表明该任务能以任何顺序实施。在线中加一个空心圆圈表示该任务是可选的，就像E/R图中的一样。这些连接方式如图11-2所示。

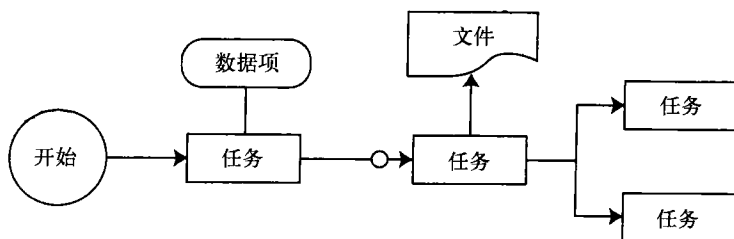


图11-2 工作过程的连接

如果你发现你的工作过程过于复杂以致不能使用这种简单的技术来描述，那么建议你使用一个在优秀的系统设计和分析书籍中描述的更完备的方法。在本书的参考目中列出了若干种选择。

11.4 用户情景

一种替代正式的工作过程分析的方法是标识用户情景。一个用户情景有两个组件组成：一是一个或多个**用户概述**，它确定预定系统中的不同用户类型；另一个是每个用户轮廓的**使用场景**，它详细描述期望用户与预定系统之间的交互方式——期望他或她实施的活动。

虽然用户情景可以用来获取与工作过程分析同样的信息，但用户情景更多地关注用户与预定系统的交互，而不是事物特定的步骤。正是由于关注与用户的交互，因此，准备那些预见系统的用户界面的用户情景一般是比较容易的。

但是，其意图是集中在用户的目标和期望上，并且正因如此，用户情景对于那些需要支持特殊活动的系统特别有用。它允许分析人员重点关注用户需要实施的任务类型，而不会陷于那些尚未定义的工作过程的构造方法。

例如，即便是简单的场景“销售代表将使用系统来跟踪他们客户订单在过程的各个阶段的状态，从初始数据录入到运送、开发票以及最终付款”，都已足够解释该用户群是如何与系统交互的。但是这么做并不意味着强制确定任何有关用户界面中的细节功能。

当然，用户情景的开发和工作过程分析并不是相互排斥的。工作过程分析是一个考虑过程本身的很有用的工具，而用户情景允许设计者关注每一类用户是如何与系统交互的。对于大多数系统来说，这些都是同等重要的问题。如果项目足够大以致需要这些话，同时实施这两种分析方法当然是值得的，特别是由于用户情景一般能建立在工作过程分析过程中所获得的信息的基础之上，而不需要额外的面谈或分析。

11.5 小结

在这一章，我们讨论了怎样理解预期系统中将要支持的活动。这可以通过实施不同程度的工作过程分析来完成，比如通过创建用户情景，或者一些技术的组合。在第12章，我们将讨论概念数据模型的设计，它是如何创建数据、结构化数据以及如何由系统使用数据的逻辑描述。

第12章 概念数据模型

在设计过程的这个阶段，你应该对你准备达成的目标有了清晰的认识。你已经定义好了项目的范围，制定了一套设计标准，并且分析好了所有的工作过程。现在可以开始构建数据模型了。

记住一个概念数据模型包含对实体、它们的属性以及它们之间联系的描述。它不是一个数据库模式，数据库模式描述的是表的物理规划。对于创建数据库模式，你还了解得不够。你需要理解用户界面和实现系统所用到的结构，然后才能创建它。

12.1 确定数据对象

在分析的前几个阶段，你已经收集或者创建了一系列的原始文档。这些包括由客户提供的文档——示例输入表格、报表等等——以及你准备的工作过程文档。创建数据模型的第一步是复查这些原始资料，并且列出所有系统需要处理的数据。

从一个工作过程开始入手。至于哪一个过程并无太大关系，但是我经常选择有关项目核心的那个，因为核心过程通常会包含大多数实体。大部分工作过程都是由一些纸张触发的，比如销售人员递交给订单录入人员一张销售订单。有时候它们也通过其他事件触发，在这种情况下第一项任务通常是填写一张表格。继续我们在第11章中订单处理的例子，一个销售订单表格的样板可能如图12-1所示。

NORTHWIND TRADERS		INVOICE				
One Portland Way, Suite 200 WA 98156 Phone: 1-206-555-1417 Fax: 1-206-555-3934		Date: 03-Nov-2004				
Ship To:	Alfreds Futterkiste Obere Str. 67 Berlin 12209 Germany	Bill To:	Alfreds Futterkiste Obere Str. 67 Berlin 12209 Germany			
10043	ALFKI	Michael Suyama	25-Aug-1997	22-Sep-1997	02-Sep-1997	Speedy Express
28	Rössle Sauerkraut	16	\$46.00	25 %	\$613.00	
39	Charlreuse verte	21	\$18.00	25 %	\$283.50	
46	Spegesild	2	\$12.00	25 %	\$18.00	
Subtotal:					\$814.50	
Freight:					\$29.46	
Total:					\$843.96	

图12-1 大多数工作过程都是由一张纸触发的，比如这个销售订单

找到这第一个数据的样板，并且记下它包含的所有信息。不要担心还没有对这些数据区分哪些是实体哪些是属性，只要记下它们就可以了。一定要记下所有重复的组，并且还要将那些工作过程分析已经确认但还缺失的数据项包含进来。最初的数据项列表可能如图12-2所示。

现在你已经编辑好列表，可以开始抽取实体、属性以及联系了。对于列表中的每一项，需要确定该项是一个对象还是对象的某些事实。对象将成为实体，而事实则成为实体的属性。这个分析的结果可能如图12-3所示。

正如你看到的，Bill To和Ship To两项被确定为归属于添加Customer实体的事实。但是，还不能立刻清楚这两个地址究竟是只属于Customer实体，还是只属于Sales Order实体，或者同时属于它们两个。区分的原则和将Unit Price包含在一个Order Detail项中一样。正如一个产品的当前售价逻辑上不同于一个特定产品实际卖出的价格一样，当前一个用户的账单地址和运送地址逻辑上也不同于该发票和货物发送的地址。

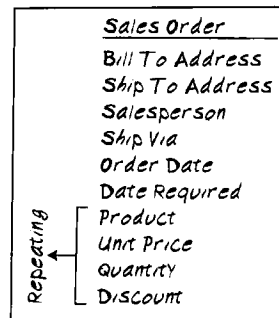


图12-2 这是销售订单表格中的数据项的初始列表

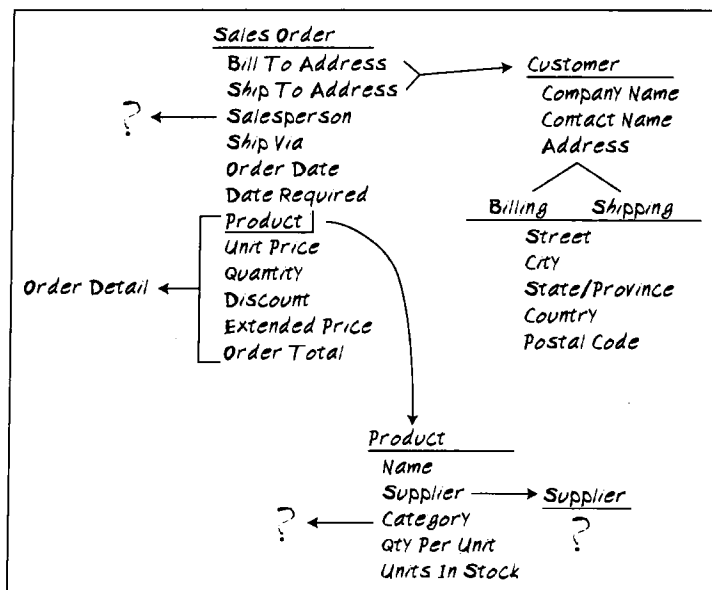


图12-3 初始化销售订单表中的实体和属性

运送地址和账单地址是否也是Customer实体的属性取决于系统的特性以及系统的用户。将这些属性包含在Customer实体中可以允许在订单输入处理过程中设定默认值，这样可以减少输入订单所需时间并减少数据输入错误的几率。然而，它也增加了在Customer实体中维护这些属性的负担。如果该企业的客户都有很多运送地址（比如，零售连锁店），那么这种负担是很繁重的。因此一个更好的解决方案就是简单地将运送地址的输入作为订单输入过程的一部分。

有时候，实行一个折中的方法会更好一些。例如，可以在Customer实体中添加多个运送地址属性，但是不要求用户填满它们。如果只有一个地址，那么系统可以使用这个地址作为默认值。如果存在多个地址，则系统提供给用户一个列表，让他们来选择。或者可以实现一些有条件的处理方式，比如使用最近输入的地址作为默认值，但是如果需要，也允许用户从列表中选择。

你还可以考虑通过订单输入过程来更新Customer实体。如果没有地址，或者一个用户输入了一个新地址，系统可以询问该地址是否应当被添加到客户记录中。这是有关用户界面的问题，并不是数据模型所关心的，但是这两者有时候联系的十分紧密，以至于不太可能将它们完全分开。

Salesperson数据项似乎表明在某个地方存在一个Employee实体，但是我们尚不知道该实体该是何种形式，因此这个问题被搁置下来。其他文档或者过程有可能会用到这个实体，如果是这样，我们可以添加需要的属性。如果不是，那么你可以决定让Salesperson作为Sales Order实体的一个属性。记住，这些决定必须都建立在系统语义的基础之上。如果数据只是为了在Sales Order中列出姓名，那么创建一个可以录入经理名、部门名以及电话分机号的Employee输入界面是完全没有价值的。

Product项已经确定为一个实体，并且在第一个图中标记为重复的项目组（Product, Unit, Quantity和Discount）将作为一个Order Details实体。从其他源文档可以推测出Product实体的一组初始的属性。从这个列表中还可以确定Supplier和（Product）Category实体，虽然还没有可用的细节信息。

由于我们是在概念层面上定义实体，因此不会在意这一点：类似Order Detail实体的Extended Price或者Product实体的Units In Stock属性是否存储为数值还是需要的时候再计算得到。我们还不知道他们事实上是否能被计算得到，但这是之后需要考虑的事情。

一项很有趣的数据是Ship Via属性。很多订购窗体都有一组为运送方式设置的复选框，可能列出诸如“Parcel Post”、“FedEx”以及“2nd Day Air”等值。这些到底是实体还是属性呢？应当依情况而定。（你也这么认为的，对吧？）首先看这里存在多少选项呢？如果选项超过两个，将它们作为一个单一的属性来构建则会十分的不便。这些选项的稳定性如何呢？可能你正在处理的是外部服务的提供商。该企业会改变提供商或者增加额外的提供商吗？企业需要对特殊的递送方式做出何种响应呢？如果一个订单要求将货物从珠穆朗玛峰山脚送到山顶，他们会拒绝这笔生意吗？如果不，那么你的模型则需要能够说明所有的这些选项。

将运送方式作为一个单独的实体来建模，可以允许这些数据项在任何时候进行更改和添加，但是其代价是数据模型和用户界面的复杂化。不同之处可能仅仅是几个按键的事情，即从各组合框中选择项目而不是用鼠标点击一个复选框。但是如果你不够细心，那些额外的按键会导致一个笨拙迟钝的界面。

如果公司必须允许特殊的递送方式，那么你需要仔细考虑怎样说明这一点。你不得不在允许足够灵活地处理所有合理的情形以及给用户强加没必要的负担之间做一个权衡。在这个例子中，最好的解决方法可能是添加一个可选的Special Instructions属性，但是这也必须在数据模型以及在任何系统过程中说明。

这些决定可能以无法预料的方式影响系统的约束。在这个例子里，虽然企业很清楚地需要知道如何将货物送给客户，但它不再只是一个要求指定运送方式的问题了。系统必须确定Shipping Method和Special Instructions属性都不能为空，这是一个需要在数据模型的不同层次中实现的有一点复杂的规则。

如果实际的运送货物也在系统范围内，那么将Shipping Method单独作为一个实体则是一个很好的想法，并且可能事实上就要求这么做，但是允许特殊的运送方式会给系统增添很重的负担。一个人如何从一个之前毫不了解的运送方式中获取运送信息呢？你可以创建一个普

通的运送实体，它包含大多数运送方式提供的属性，比如一个标签号和收取时间，或者可以指定已知的运送方式，并且将特殊的方式作为在系统范围外处理的例外情况。

这里存在的问题就是系统过于复杂并且给用户添加了不必要的负担。确实很容易对系统能够提供的功能感到兴奋，但却忘记了可能会造成的负担。的确，提供默认值是一件好事情，前提是它们易于维护并且这种维护是常规的（最好是作为某些其他任务的副产品）。可能的话，最好让接待人员来处理运送要求，但是为了满足五位客户的要求，就为一千个订单输入运送细节信息，这样值得吗？

这些决定仅仅要求你思考做出的设计计划。但是很容易在你第一个闪过的念头中将其忽略，“这不是很酷吗？它会给我们节省很多时间”。当你在系统的某个地方获取一些数据后，你需要考虑它是否会在系统的其他地方用到，还是仅仅提供了一个默认值或者作为一种约束。如果你正在以某种方式输入运送细节信息，那么为什么让它们对接待人员也可用呢？

相反地，不论你在哪使用数据，都应当考虑它是在哪里生成的以及它如何维护。作为一个通用规则，最好是提供给用户一个选择列表而不是一个文本框。但是在创建和维护列表，以及在构建维护界面中都需要一定的成本。当对你的数据模型的结构做决策的时候，所有的这些事情都需要权衡好。

当然，你的目标不会是要要求一个数据被输入两次。但是同样来说，你也不希望强迫用户在他们使用方式之外的某处输入数据，以便在他们想要完成的任务中使用它。我们会在第四部分详细讨论这个问题，但是确定数据是在哪生成的以及在哪里使用是这个过程中最关键的第一步。

12.2 定义联系

在浏览完毕所有的源文档后，你会对问题域中的实体及属性有一个大致的描述。还剩下两个任务：建立这些实体之间的联系，以及复查每一个实体的属性和约束。

虽然理论上，你可以先检查属性，但是我发现从联系开始会更简单，因为其中的一些可能成为额外的实体而一些可能要求给已经确定的实体添加属性。

如果你和我一样，那么在你第一次浏览完所有源文档之后，应当有一些手写的笔记，它们是一些箭头、涂鸦以及“请看第12页”的字样，很可能没有其他人能够解读得了。因此，定义联系的第一步就是将这些草稿整理清楚。然后就可以开始构建数据模型的第一个实体联系（E/R）图草稿了。（如果你的笔记确实太乱并且你担心即便是你也要三个星期才能读懂它们，那么你也可以为每一个实体列出你已经确定的属性。）

从选择一个实体开始，通常是系统中的核心实体之一，然后添加与它有联系的其他实体。你可以在这个过程中定义该联系的类型（一对一、一对多、多对多），或者只简单地画一个直线来提醒你这里有一个联系，之后回过头再分析它。我通常会在这个过程中分析，但是你可能会发现先把所有实体记下，然后再复查它们的方式会更容易一些。

第一个订单-处理例子的E/R图的草稿如图12-4所示。这是一个样本示例，并且这个图很好读懂。（假设我们已经决定Salesperson仅仅是Sales Order实体的一个属性，而不是将其自身作为一个实体。）如果你正在实施一个复杂的例子，那么可能想要创建多个图形，每一个都只描述数据的一个子集。在这种情况下，最好使用支持图形的自动工具。否则，保证它们的同步会是十分繁重的工作。

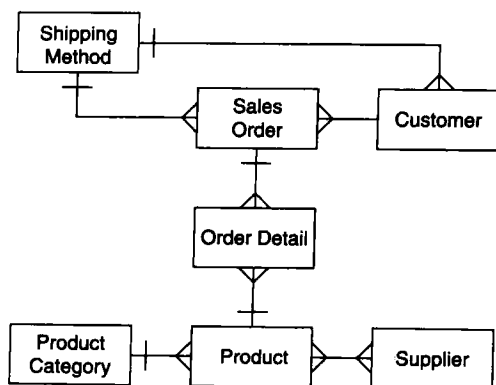


图12-4 订单处理E/R图的第一份草稿

一旦已经准备好E/R草图，就可以开始更细节地分析这些联系。对于每一个联系，需要决定以下事宜：

- 联系的基数
- 每一个参与者的可选性
- 联系的任何属性
- 联系的任何约束

图12-5显示了复查之后的订单处理E/R图。

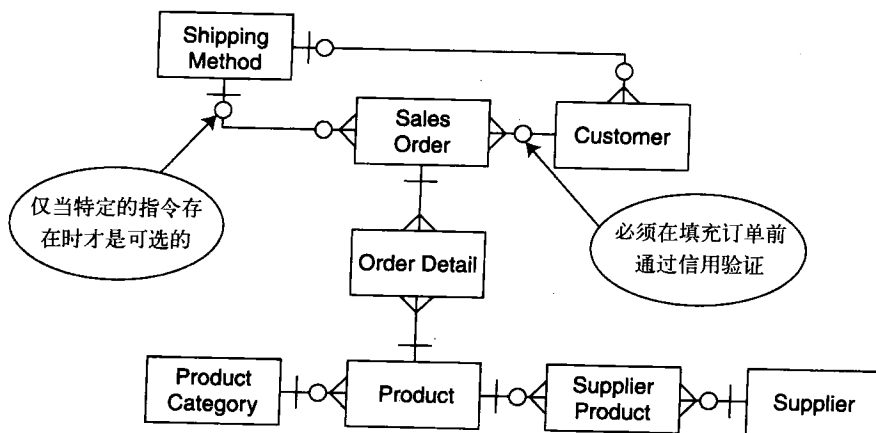


图12-5 在复查联系之后的订单处理E/R图

12.2.1 联系的基数

可能在你的第一份草稿中已经表明了实体之间的联系，正如我们在图12-4中所作的。如果没有，现在正是这么做的时候。即便你已经完成了，最好现在也复查你的决定以便对整个模型能有一个更完整的了解。

在所有多对多联系的地方，应该在模型中添加一个连接实体，使得每一边都是一个一对多联系。在我们的模型中，Supplier和Product之间的联系就是多对多的，因此我们需要添加一个Product Supplier实体来解决这个问题。注意Sales Order和Product之间的联系也是多对多的，

但是这个情况中是Order Detail实体充当连接的功能。

12.2.2 联系的可选性

已经在任意两个实体之间建立了联系的种类之后，现在应当考虑该联系中的某个参与者或者两个参与者是否是可选的。在我们的例子中，Customer和Shipping Method之间的联系在两个方向都是可选的——这就是说，客户并不要求一定要有一个默认的运送方式，并且运送方式可以脱离客户独立存在。

然而，Product Category和Product之间的联系只在一个方向是可选的。一个产品类别不一定要有归属于它的产品，但所有的产品必须要属于某个产品类别。

Sales Order和Shipping Method之间的联系更加复杂。一个运送方式可以独立于销售订单而存在，因此该联系的Sales Order端是可选的。但是，联系的Shipping Method端仅仅只在销售订单有特殊说明时才是可选的。这是一个很重要的约束，应当在图形中标示出来。

12.2.3 联系的属性

在大多数情况下，对于实体间的联系所需要记录的就是它存在的事实。例如，我们需要知道一个给定销售订单的特定客户，并且这是我们需要知道的全部信息。然而，有时候我们有必要知道联系的其他事实——比如，它什么没时候开始以及将持续多久。这些事实是联系本身的属性，而不是属于哪一个参与者的。

当联系本身拥有属性时，它必须单独构建为一个实体。在订单处理的例子中，我们可能决定指定一个Supplier拥有“Preferred Supplier”的状态。由于在Product和Supplier之间已经有一个连接实体，那么这个Preferred Supplier属性就可以简单地添加给这个实体。如果不是这种情况，那么我们就需要添加一个实体来描述联系的这些属性。

12.2.4 联系的附加约束

最后，我们要考虑该联系是否有任何附加的约束。比如，一个一对多联系的“多”端最少或者最多能存在多少条记录？在联系允许存在之前是否有任何条件？是否存在联系必须存在的任何条件？

在我们的例子中，Sales Order和Shipping Method之间联系的那个要求——只有当指明Special Instructions的时候，它才是可选的——就是一个这样的约束。另外，客户只有在他们的信用认证后才能下订单的规则也是一个约束。同样，这条规则也需要在图形中标明出来。如果存在很多约束或者这些约束对于简单的标示显得过于复杂了，就可能需要在其他地方记录它们，至少应当在图中表明该约束的存在。

12.3 复查实体

现在你已经获得了系统中实体的全貌以及它们的连接方式，是开始细节分析每一个实体的时候了。对每一个实体，需要确定下列信息：

- 实体和问题域之间的联系
- 创建、修改、使用和删除实体的工作过程
- 它可能交互或者依赖的其他实体

- 属于该实体的业务规则和约束
- 实体的属性

12.3.1 实体和问题域之间的联系

确定实体和问题域之间的联系通常是很简单的。“Customer实体是购买我们产品的个人和企业的模型”。我认为这里最大的问题是想出一句不那么赘述的话。比如，“Employees实体是对企业的雇员的建模”似乎就不值得说。

如果一个联系是作为一个实体来建模的，那么事情会变得很麻烦，因为实体并不直接映射到问题域上。“一个供应商可以提供多种产品，并且任何给定的产品可能由任意数量的供应商来提供。Product Supplier实体构建了这个联系，同样还为一个特定的产品的任何给定的Supplier构建了Preferred Supplier状态。”

问题域中的有些事物——Sales Order很可能是最好的例子——是使用数据模型中的一个或多个逻辑实体来建模的。我们将这样的实体称为**复合实体**（composite entity）。销售订单文档同时由Sales Order和Order Detail实体来描述。

一般来说，针对建档目的，将复合实体作为一个单一的对象来处理会更加清楚。例如，“Sales Order和Order Detail实体描述了一个客户的一个单一的订单。Sales Order实体对订单本身进行建模，而Order Detail项目描述了每一个被订购的产品”。

12.3.2 影响实体的工作过程

虽然你可能已经确定在之前实施的工作过程分析中数据项的使用位置，但在实体文档中包含这些信息仍然是很有用的。这样，如果在某个时候有必要对实体的结构作某些改动，比如增加一个属性，那么只在一个地方就可以确定受影响的所有过程。

确定直接操作在一个实体上的过程通常也是一个简单的过程，而确定那些不直接与实体交互的过程可能就需要花些功夫了。例如，可能不是很明显，订单实体过程可以修改一个客户的默认运送方式，或者给某个产品类别确定的一个“特别红利”可能影响到折扣以致整个销售订单的总额。如果这些信息没有被仔细地记录下来，那么这种类型的交互将会成为一名维护程序员的噩梦。

大多数分析人员会将这些在工作过程分析中的交互信息记录下来，当需要变动过程本身的时候，这会十分有用。然而，有时候模型本身会发生变动，这可能是由于业务环境的一个改变而发生的直接变动，也可能是由于模型中已存在的过程需要改动而发生的非直接的变动。在这种情况下，浏览整个实体文档来查看你要变动的特定实体会比筛查所有工作过程来确定那些会受变动影响的实体要容易得多。

考虑一下将工作过程的信息包含在你的实体文档中，它可以作为交叉引用来使用。和所有的交叉引用一样，它很难实现和维护，但是从长远来看会使得你的生活轻松很多。

12.3.3 实体间的交互

E/R图是很优秀的工具，但是它们只能显示有限的信息。如果系统中的实体有很复杂的交互但不能很简单地在图中描述出来，那么将它们记录在实体描述中是十分重要的。即使你已经在图中添加了标记，你也应当详细描述任何没有直接出现在标记中的交互。

如果该模型过于复杂以致于存在多个实体图形，并且一个给定的实体出现在多张图中，那么在实体描述中列出所有与它联系的实体会很有意义。这类似于在多处提供查询值的实体的情况。例如，一个Courtesy Title包含的条目有“Mr.”、“Mrs.”、“Dr.”以及“Ms.”，它可能会在多个地方被引用。如果需要对该实体作任何变动，那么最好是能够在一个单一个地方找到所有有关的实体。

然而，作为一般规则，E/R图为实体间的交互提供了足够的文档。只有一些类似之前引用的例外情况会要求额外信息。

12.3.4 业务规则和约束

对一个实体来说，下一部分文档就是记录属于它的任何实体级别的约束。任何引用多个属性的约束，比如在我们的例子中，“Shipping Method和Special Instructions属性不能都为空值”这条规则，也应当记录下来，并且现在正是这么做的时候。

12.3.5 属性

实体需要的最后一部分文档就是属性列表以及它们的域。在编辑列表时，需要从你在浏览源文档时已经确定的属性列表开始，然后确保添加了表达参照完整性所需要的外码。

还要检查每个实体至少有一个候选码，它是用来唯一确定每一个实例的。这将成为数据库模式中表的主码。记住主码不能包含Null值。由于这一点，并不是总是可以使用现成的一个属性或者多个属性的组合来作为主码。如果是这种情况，则需要添加一个随机数，即系统生成的标识符。

在我们的例子中，Customer实体很可能有一个人工的标识符。如果我们假定一个客户可能要么是一个个人或者一个公司，那么你会有一个类似图12-6所示的初始的属列表。

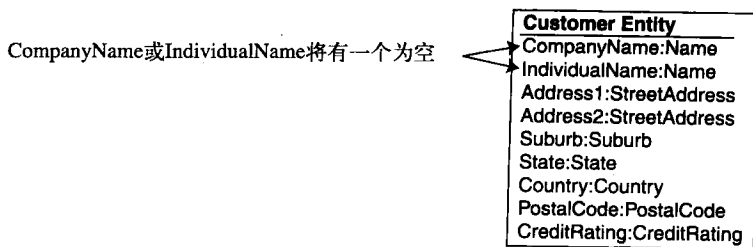


图12-6 Customer实体属性列表

即便我们不考虑名字不唯一的问题，它依旧存在问题。如果该客户是个人，那么Company Name将会为空。如果该客户是一个公司，那么Individual Name将会为空。总是有一个将会为空值，即使我们假设它们都能唯一的确定该记录，这些字段也不能被用作候选码，事实上我们还不能这样假设。

现在我们来讨论有关Customer实体的下一个问题——名字不唯一。在我们的例子中，整个属性列表甚至都不能保证是唯一的，因为有可能两个客户拥有相同的名字并且住在同一个地方。John Smith不会告诉你他实际上是John Smith Jr. 并且不会与他爷爷的名字相混淆，而他的爷爷居住在同一所房子里。

John Smith Sr. 和 John Smith Jr. 是不同的两个人，并且应当用不同的记录区分开来，这是毫无疑问的。但是，唯一确定他们的这些属性并不是我们的业务范畴。你可以想象一下，当你在订购日用品的时候却被询问你的居住安排情况。“抱歉，先生，我想确认一下您是否凑巧有某位亲人和您住在一起并且名字相同？这是为了我们的计算机系统而做的检查。”即便是在最传统的客户服务，也不可能会有这样的情况发生。

幸好有一个简单的解决方法：客户代码（Customer Number）。并且，即使企业尚未存在一种分配该代码的方法，Microsoft Jet 和 Microsoft SQL Server 都提供了随机分配代码的机制。（分别是，AutoNumber 和 Identity 数据类型）。

即使使用了随机数标识符，也一定要确保提供一种替换标识的方式。你也不愿让用户由于忘记了他或她的客户代码而不得不拒绝订购吧。比如，一种方式是询问客户，“您是居住在 Oakridge 还是 Cincinnati 的 John Smith？”，而告诉他找到过去的某张发票后打电话过来则完全是另一种方法。

Customer 实体也是使用随机系统标识符的第二个原因的一个例子。即使我们可以假定名字和地址的组合对于我们的目的来说足够唯一了，但是仍旧会导致太多的字段在多处进行复制。记住一个实体的主码将会在任意某个实体中作为外码来使用。显然复制单个属性会比复制五、六个属性要高效的多。

12.4 域分析

在图12-6中，列出的属性采用了“名字：域”的形式。很多分析者都忽略了域的存在而直接以数据类型和约束的形式来指定属性。不过，即使你忽略了过程中的这一步骤也没有问题。这可能不够正确，但并不可能所有人都会因此而责怪你。

在我的工作中，我要实施域分析的原因是它会减少工作量并且会提供额外的信息，因此我也推荐你这么。我认为，越简单越正确的方法就是好方法。而且这项任务还有额外的技术上的优点。

让我们看一个例子：图12-6中的 CompanyName 和 IndividualName 属性表明它们的值都是从 Name 域派生出来的。

现在我们可以按如下形式定义 Name 域：

“由一个或多个词以恰当形式组成的字符串，它的最大长度为75个字符，只允许有字母和标点符号句号（.）和逗号（,）。”

我们只能定义一次该域，而可以在整个系统中多次引用它。我们本应该为每一个应用的属性定义这些约束的，但为什么这么麻烦呢？进一步来说，由于这些属性是在同一个域上定义的，因此它们在逻辑上是可以比较的。但如果我们直接定义了这些属性的话，这种可比性就不那么明显了。

在 CompanyName 字段查找和 IndividualName 字段有相同的值的记录可能并无太大作用，但是这至少是一件可能的事情。将公司名称和客户代码相比较，不可能说它们凑巧有相同的结构和约束。

从技术上来讲，一个域的定义是“一个值的集合，属性可以从中派生出它的值”。这从概念上是很简单的，但是到底如何定义一个域呢？本质上来说，需要确定三件事：

1. 域的数据类型

2. 数据类型允许的取值范围内的任何限制
3. 属于域的任何格式, 该项是可选的

选择数据类型

定义域的第一步就是选择核心的数据类型, 它会在数据库模式中被描述。这也是违背关于数据库模式和数据概念模型相分离的原则的一个例子。

该数据类型起到了简要描述值范围的作用。除非你是在对数学进行建模, 否则“Integer”就不是一个域, 域“Quantity”的值几乎可以肯定是整型的。然而, 我并不推荐过多的着重于具体的数据库引擎类型。站在这个角度来说, 数据库引擎的选择仍旧是可以改变的。

一个域的“数据类型”也可能是另一个域的。你可以定义一个通用的Date域, 例如它可以指定系统中的所有日期都必须不早于1900年1月1日并且使用一个四位数字的格式。定义EventDate为“1982年10月23日之后的一个日期(交易开始的日子)”是完全可以接受的。

12.5 限制值的范围

在为域定义了数据类型之后, 下一步就是指定对于该域有效的在该数据类型范围内的值。有时候, 最简单的方法是指定一个规则: “数量必须是正整数”。

有时较简便的方法是列举一个域的所有有效值。“地区必须是以下几种中的一个: 西北、东北、中部和南部”。在这个例子中, 你几乎肯定想要将该域作为一个实体包含在数据模型中。这样会更加简单, 用不着在它们被引用的所有地方都输入那些数值, 并且也允许它们在系统实现后很容易地改变。

这项规则唯一可能的例外情况是当域的值数量太少并且不可能被改变。比如, 你正在对一个调查问卷和一份考卷进行建模, 并且你有一个Answer域, 它由“True”和“False”两个值组成。那么将这两个选项作为一个实体来建模就没有意义了。这里不存在其他可能的值, 因此在实现过程中引用一个表肯定会比直接在规则中输入要麻烦很多。

你也要用一个实体对那些必须使用多个属性的域建模。对此最好的例子是State域。如果必须说明多个国家, 那么就不能在不引用特定国家的情况下, 决定一个给定的州的值是否有效。

例如, 如果一个客户住在澳大利亚, 那么“New South Wales”就是一个有效的州名, 但是“Alabama”则不是。在这种情况下, 域的查找实体将同时由Country和State属性组成。这个例子并不是严格意义上的域定义, 并且它是通过E/R模型中要求的联系来建模的。然而, 将这种情况认为是一个种复合域会简单些, 并且就按复合域来处理它。

毕竟, 这里的关键是简化确定系统约束的任务, 并且为重复出现在数据模型中的域绑定域定义将会节省时间并减少出错的几率。

域定义还需要说明定义在域上的属性的值是否接受空值或者零长度的字符串。即使是在使用一个系统实体在对该范围进行建模, 在定义中清楚地声明这一点是十分有用的, 在这种情况下, 是否允许为空就由两个实体间的联系来决定了。

实施域分析并对一个给定的实体确定属性列表是联系紧密、反复迭代的过程。在实际应用中, 你会发现在列举属性的同时定义域是十分有效的。如果一个属性的域已经被定义了, 那么可以简单地罗列它。如果没有, 那么可以在有一个例子的时候定义这个域。

在这个过程中, 你可能会发现某些属性除了定义在域上的约束之外, 还有其他的限制。

这是完全可能的，并且也并非罕见情况。例如，你可能已经定义了一个Event Date域，它表达任何事件可能发生的日期。该日期被限制在公司开始交易之后的日期。在Sales Order示例中，Order Date和Shipping Date两者都可以定义在Event Date域上。但是，Shipping Date属性还必须是在Order Date之后。这是一个实体级别的约束并且应当在实体描述中列出。

在定义域约束的过程中（就此而言，还有附加的属性约束），应当尽可能详尽，并且不影响可用性。我们将在第四部分更加详细的讨论这一点，但是在此，你应该意识到定义的域越精确，给用户提供的帮助就越大。但是，如果你偶然的省略了某些值，那么这将会妨碍用户的使用并最终导致系统的不可用。

定义格式

这并不是绝对必要的，但是通常给一个域指定合适的格式是一个很有效的方法。如果你一旦指定了所有的日期都必须显示为DD-MMM-YYYY的格式，那么你以后就再也不需要做这些了。

12.6 规范化

这可能有一点意外，没有其他任何地方在讨论数据建模的时候像我这样讨论规范化数据模型。这是我个人的经验之谈，如果你由零碎的数据开始，然后将它们组织成各个实体，协调重复组和多对多联系，如此这样的进行下去，你很可能就得到了一个三范式形式的数据模型。

但是它的确没有什么坏处，特别是当所有这些对你来说还很新颖的时候，为了一致性要复查模型。记住，模型中的每一个实体都应当依赖“主码，主码，除了主码，别无其他”的原则。

12.7 小结

本章，我们讨论了构建系统的概念数据模型。这个过程开始于检查所有源资料以确定系统使用的数据项，然后将它们组织成一系列的实体。接着审查实体间的联系，然后需要分析每个实体以及它们的属性。

在下一章，我们将探讨将概念数据模型转化为物理数据库模式，它将针对所选的数据库引擎进行实现。

第13章 数据库模式

在上一章，我们学习了概念数据模型，它定义了数据的逻辑结构。本章，我们将转向数据库模式，它描述的是数据的物理结构。请记住数据库模式仍然是一个逻辑结构。在构建它的过程中，你要在相当抽象地层面来描述数据的物理结构，尽管它对应的是数据的物理表达。实际的物理描述是数据库引擎的责任而你无需关心。

13.1 系统架构

在描述数据库模式之前，必须先确定系统需要的架构。不幸的是，很多著作使用术语“架构”来描述两个绝然不同的模型（虽然有关联）。为了澄清这些事情，我们将称其中一种模型为**编码架构**，而另一个称为**数据架构**。但是请注意这些是我命名的，你不太可能在其他地方找到它们。

13.1.1 编码架构

我所称的编码架构在其他不同的著作里被叫做“应用模型”（application model）、“层次范例”（layered paradigm）以及“服务模型”（services model）。编码架构描述的是代码的逻辑组织方式。编码结构绝大部分是一个实现问题，并且正因如此，它不属于本书的范畴。然而，编码架构可能影响到是否在数据库模式中实现了数据的完整性约束，因此我们在此要讨论它，虽然有一点多余。

在过去十分糟糕的日子里，系统架构是一个单一的整体：大量的无结构的代码。任何人不幸如果试图要修改（或者甚至理解）一个稍微复杂一点的系统，就好比他们不可能再次看到盘子里相同样子的意大利面条一样。为了在这种混乱情况上强加某些次序，程序员们开始将他们的代码以不同的形式组织成离散的组件：子程序、模块或者对象，这依据编程语言的能力。该方法的问题是虽然不是意大利面条，取而代之的是方便的意大利式饺子——独立的代码段之间以某种方式进行交互，但是没人能知道是如何交互的。

为了管理这种现代的意大利面食，很多开发者正将组件组织成各种服务，有时候称作“层”，它们在离散的逻辑层面实施任务。组织层的方式有几种。我们将讨论两种最常见的形式：三层模型和四层模型。

13.1.1.1 三层模型

三层模型将各个组件组织成用户服务层、业务服务层，以及数据服务层。给用户描述信息以及响应用户动作的编码组件被指定为用户服务层。整个用户界面在这一层中被封装起来。业务服务层负责强制执行的业务规则以及保证用户输入的有效性。业务服务组件和用户服务层以及数据服务层都要相互交互。数据服务层中的编码组件只与业务服务层进行交互，它负责维护数据。

这个三层模型十分清晰，但是我发现在实际开发中它是存在问题的。似乎总是存在某些特定的功能类型是不属于任何一个层次的。比如一些数据在显示给用户前需要格式化。一个

社会保障编号可能被存储为一个9位的字符串，但是显示的格式却为999-99-9999。这个格式是属于用户服务还是数据服务呢？你可能将它归属于其中的任意某个层次。同样的，事务管理是业务服务还是数据服务的一部分呢？当开始使用层次数据和数据修整来设计复杂系统时，这些类型的决定会变得十分繁杂。

除非你是非常一致的，我想你将这种类型的功能置于何处都是无关紧要的，否则这一定是该模型失败之处。如果你必须要引用一套额外的规则——比如是“格式应当属于用户服务层，并且构建层次数据集合是业务服务层的一部分”——那么，该模型的负担就要超过它所带来的收益了。

13.1.1.2 四层模型

将编码架构分解成四层而不是三层，能够消除很多三层模型所附带的问题。这个四层模型经常被称为“层次范例”，它将代码组件组织成用户界面层、数据界面层、事务界面层以及外部访问界面层，如图13-1所示。

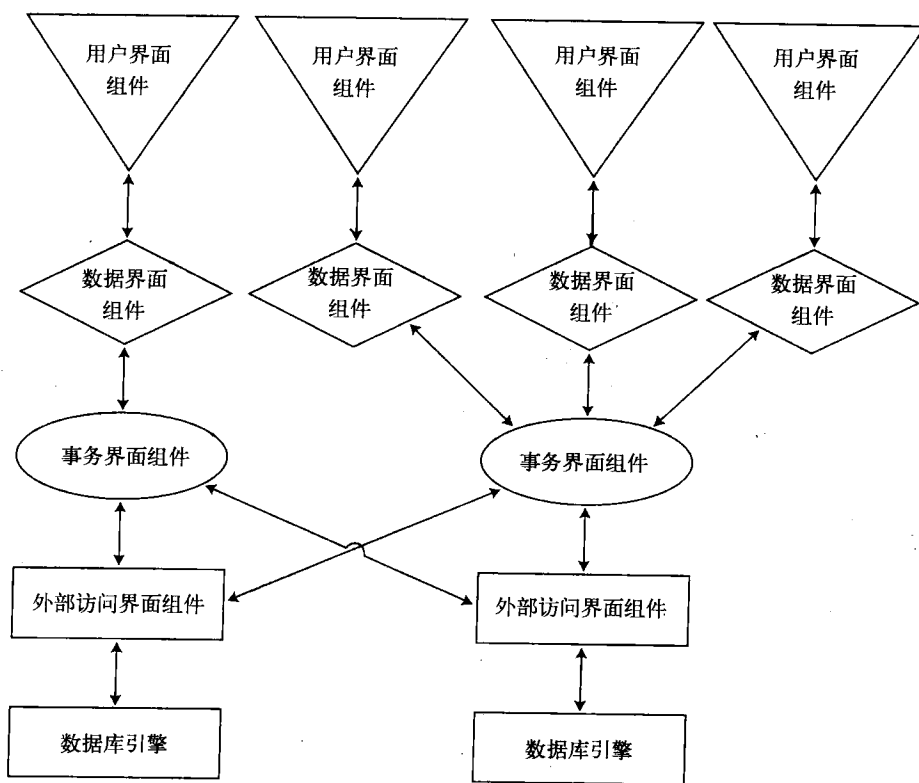


图13-1 四层模型

其中，用户界面层对应的是三层模型中的用户服务层。用户界面层负责与用户的交互，包括通过窗体对象给用户表达信息；响应窗体对象状态的变化，例如一个窗体大小的改变；以及初始化用户需求。

数据界面层负责在内存中维护数据（并不是永久性的维护它，这是外部访问界面层和数据库引擎的工作，这一点我们马上就会看到）。它显式地包含了大多数在三层模型中存在争议

的功能，比如格式化数据并且创建虚拟记录集。（一个虚拟记录集仅仅存在于内存中，它不在任何地方永久存储。）

在大多数情况下，一个数据界面层的组件与用户界面层的一个特定组件是紧紧组合在一起的。但是，理论上，一个数据界面组件可以支持多个用户界面组件，如图13-1所示。例如，一个系统可能包括一个客户维护窗体来显示单个客户的信息，还包括一个客户总结窗体来显示多个客户的信息。由于这两个窗体都描述的是一个客户实体，因此它们可能共享格式化和有效检查CustomerNumber的代码，这是数据界面层的一项功能。

在物理层面上，一个用户界面组件通常对应的是Microsoft Visual Basic或者Microsoft Access中的一个窗体，并且相关联的数据界面组件通常会在窗体的类模块中实现。然而，某些过程可能是多个窗体共享的，在这种情况下，这些过程需要在一个共享模块中实现。

数据界面层负责数据有效性的检验，但是并不对业务过程负责。例如，系统需要确保在一个订单中指定的CustomerNumber值对于系统来说必须是已知的，这样的功能代码就是数据界面层的一部分。而那些强制事件特定次序的代码则属于事务界面层，比如防止订单在确认客户信用前发出等。

事务界面层通过应用程序协调数据的使用。这一层的组件负责构建和初始化查询，从外部访问层获取信息，强制业务过程，以及通过外部访问界面层处理错误和报告冲突。

事务界面层的组件比数据界面层的组件相更易于重用。对于Visual Basic和C#来说，事务界面层组件十分利于作为实现对象。例如，一个Customer对象可能提供一个Update方法，这个方法被多个数据界面层的组件调用。注意，由于数据界面层应当（至少理想上）与用户界面层的组件是相互独立的，因此，需要更新的值必须被显式地列出来。换句话说，该调用将采用下列形式：

```
MyCustomer.Update CustomerNumber, CustomerName ...
```

这个Update方法之后将创建UPDATE查询语句，并把它提交给外部访问界面层来执行，并且处理可能带来的任何出错情况，或者是直接解决这些错误，也可能将它们提交给用户界面层显示出来。

外部访问界面层负责应用程序和外部数据源之间的交互。在数据库系统中，这一层的代码组件处理与数据库引擎的交互。它们执行这些查询并将查询结果（包括任何错误信息）返回给组件。

理想的情况下，应该在这一层设计子过程将事务从你选择的特定数据库引擎中独立出来。理论上，仅仅通过替换外部访问界面层，而将原本设计使用Microsoft Jet数据库引擎的应用程序扩充成使用SQL Server是可能的。但实际上，这实现起来可能就有一点困难了。

请记住，事务界面层是负责构造查询的，而外部访问界面层仅仅执行这些查询。如果SQL语言的实现之间存在语法差异的话，实现起来就不那么简单了。一个由Microsoft Jet数据库的TRANSFORM查询创建的记录集，如果使用SQL Server则需要创建多条语句才能实现。

如果能够提前预知由应用程序执行的所有查询，那么就可以避免由于在数据库模式中包含查询而造成的语法问题。在这里，参数查询可能会十分有用。你不必马上创建一个SQL语句来查询名为Jones的客户——你可以只将“Jones”作为参数传给一个预先存在的查询，这样可以简化你的代码架构并且还可能提高性能。

不幸的是，并不是总能提前预知所有可能的查询，特别是你为用户提供了一种特殊的查

询能力的时候。在这种情况下，几乎不可能将事务界面层整个独立出来。（至少，我尚未找到该问题的一个完美的解决方法，如果你找到了，我将十分感激你能告之于我。）同时，你可能不得不在数据界面层组件中编写一些条件代码。

如果你的应用程序被设计成仅仅支持一种数据库引擎，那么可以尝试将事务界面层和外部访问界面层融合在一块。但我并不推荐这么做。虽然设计外部访问界面层将耗费一些时间，但是这个过程并不困难，并且一旦完成，它将在系统的其他地方为你节省数百行代码。进一步来说，一旦完成了一个通信的组件，比如在SQL Server 2000环境下使用ADO.NET，那么你就再也不需要重复写它了。你可以在任何你写的其他系统中使用它，而不需要修改。只有当底层的数据库引擎或者对象模型改变时，才需要一个新的外部访问界面层。

13.1.1.3 代码架构和数据库模式

你选择的代码架构将对数据库的两个方面产生影响：外部访问界面层（或者是数据服务层，如果你使用的是三层模型）的独立性以及数据的有效性。我们已经讨论过通过预测必要的查询并将它们包含在数据库模式中的方法，使得外部访问界面层独立于数据库引擎的改变。这个方法还有额外的提高性能的优点，有时候这个提高是十分明显的。数据的有效性检查在某种程度上是一个十分困难的问题。我们将在第19章更详细的探讨“什么是数据有效性”以及“如何进行有效性检查”。在这里我们只讨论“何时”以及“何处”进行数据的有效性检查。

一些设计者鼓吹要将所有的数据有效性检查的功能都嵌入到数据库引擎自身当中。这个方法不是没有好处：所有的数据完整性约束以及业务规则都在一个地方实现，这样它们可以很容易地被更新。不幸的是，这个方法也存在问题。

首先，某些规则不能在数据库引擎一级实现。例如，在没有触发器的Jet数据库引擎中，一旦记录创建，就很难强加一个规则来阻止主码值发生改变。即使在SQL Server中，虽然这方面显得更加有效，但你也不可能在数据库引擎里直接实现所有的规则。

YUKON注意：SQL Server Yukon允许在任何过程语言中实现触发器、代码和存储过程，这样可以消除在服务器级别实现约束的任何有效限制。

其次，在数据提交给数据库引擎之后才能进行有效性检查必然会降低系统的可用性。通常的方法是一旦数据输入就应该进行有效性检查。在一些情况下，这意味着一旦按下一个键，就进行有效性检查，比如阻止在一个数字字段输入字母字符。在另一些情况下，应当在退出某个字段或者在输入完字段的最后一个符号后检查数据的有效性，比如实行一条规则：DesiredDeliveryDate值必须等于或者晚于OrderDate值。

哪怕是在一个单机上运行的一个独立的应用程序，在每一次按键后或者是在每个字段退出时，都将一个数据集合提交给数据库引擎会导致很糟糕的性能。如果你是在网络上做一个来回的访问，或者（但愿不会如此）通过一个宽带网络或者Internet来访问在一个远程站点的数据库引擎，性能会十分低下以至你的用户不使用这个软件，而换为使用索引卡（并且它们可用）。

如果所有的有效性检查都由服务器处理，那么唯一的解决方法就是在整个记录都完成以后再将数据提交给数据库引擎作有效性检查。但是这样一来，用户的注意力将会转向下一个问题。显然，报告10分钟前的一次输入所造成的错误会让人觉得混乱和迷惑。

然后，为了使系统尽可能的快速和可用，必须在应用程序中实现数据有效性的检查。如

果数据库只被一个应用程序使用,并且有效性需求相对来说是稳定的,则可以决定只在应用程序层实现数据有效性检查,而且这些检查完全在数据库引擎进行有效性检查之前。这消除了重复的工作,但这是一个相当危险的方法。

如果将来有另一个应用程序要在相同的数据库上实现,那么只有好的设计意图可以防止新的应用程序破坏该数据库的完整性,并且我们都清楚好的设计意图用在哪些方面。即使数据库不会被另外的应用程序共享,让用户使用特殊的工具,比如Access或者SQL Server企业管理器进行数据操作也是很很不安全的。一个严格安全的模型除了应用程序本身之外,严禁从其他地方修改数据,但是如果以限制访问数据为代价是不太合适的。

鉴于这些原因,我们相信最好的方法是在应用程序和数据库模式中都实现数据有效性检查。Access自动实现了这一点。如果你在表级别为一个字段定义了一个有效性规则,那么之后当你将该字段拖到一个绑定窗体上时,该窗体将会继承这条有效性规则。

不幸的是,在Access 2000 之前的版本中,这项功能同时还是复制有效性问题的一个典型例子。如果你将该字段包含在一个绑定窗体上之后,你改变了这个表级别的有效性规则,那么这些变动是不会反映到该窗体上的。Microsoft Access 2000改进了这条规则,但是在Visual Basic中依然存在这样的问题。如果在多个窗体中(可能在一个或者多个应用程序中)引用该字段,那么必须在每一个窗体上手动地修改这些有效性规则(或者,在数据界面层组件中支持每一个窗体)。

为了解决这个问题,可以在运行时查询数据库引擎的有效性规则。该技术会给系统造成一定的负担,但是如果你的有效性规则频繁变动的话,这项负担可能会通过在一个地方更新这些规则所带来的便利性抵消掉。

可以在应用程序开始的时候,或者一个窗体加载的时候,以及在每条记录更新之前从数据库引擎中获取有效性规则。我建议在窗体加载的时候做这些。如果在应用程序开始的时候获取有效性规则,你可能得到一些属于用户尚未加载的窗体的不必要的信息。如果你在每条记录更新之前这么做,那么可以保证你正在操作的这些规则就是当前最新的规则,但是这意味着对每条记录实行重复查询,实际上,有多少系统是如此不稳定的呢?

存在一种极端不可能的情况,就是当用户打开一个窗体的同时某条规则发生了改变,并且用户输入的数据通过旧的规则编译后与新的规则发生了冲突,这个问题无论如何都会被数据库引擎捕获,因此不会造成很大的破坏。此外,在系统正在使用的时候,考虑弥补数据库模式就已经足够让我头疼了。

13.1.2 数据架构

除了决定系统的代码如何构成之外,还必须确定一个数据架构。你可能会想起在第1章中,一个数据库系统由若干离散的组件构成:应用程序本身、数据库引擎以及数据库。(请查看图1-1。)在四层代码模型的基础上,我们可以在某种程度上重新定义这一结构,如图13-2所示。

在为应用程序决定数据架构的时候,你需要确定每一个所在的层次。理论上,每一层(或者甚至是每一个组件)可以存在于不同的计算机上,通过网络之类的途径进行交互。另一方面,所有的组件也能都存在于一个单一的计算机上。但实际上,一些相对标准的配置已经针对不同的情况开发出来了,我们将依次讨论它们。

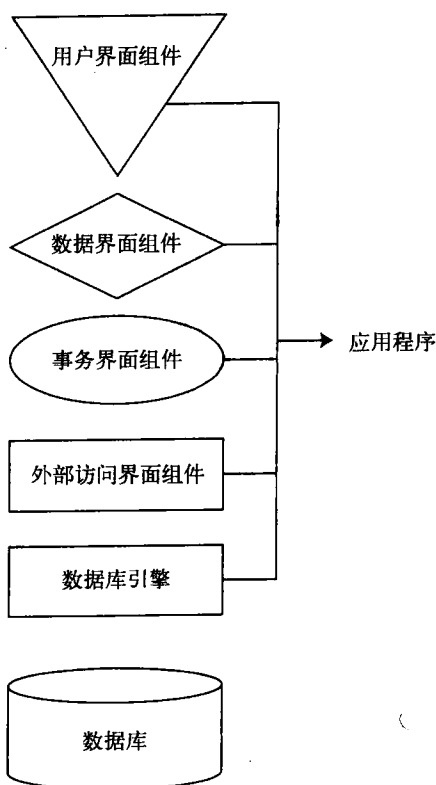


图13-2 一个数据库系统由六个离散的层组成

13.1.2.1 单层架构

数据架构中的每个逻辑组件组都对应着一层。自然，最简单的架构就是一个单层系统，这样的系统里面所有的组件都处在单个逻辑层上，并且最简单的一个单层数据架构的形式就是一个单机系统。

在一个单机系统中，所有的存在于该单机上的组件都只对实际操作该机器的用户可用。虽然该机器可能偶尔连接到某个网络或者Internet，但该数据库系统并不对其他用户开放。由于所有的处理都只在该机器上进行，并且数据的存储也是本地的，所以这台机器的性能约束就是这台机器的能力——它的处理器速度和内存。单机系统会显得十分缺乏内存，这也是寻求其他架构的原因之一。

绝大多数的单机系统都使用Jet数据库引擎，虽然Microsoft一直鼓励开发者使用Microsoft Data Engine (MSDE)，它是作为Microsoft Data Access Components (MDAC) 的SQL Server 单机版发布的。它当然可以在一个单机上实现一个SQL Server系统，但是鉴于某些原因（之后将详细阐明），它是否符合一个单层系统的要求尚存在争议。

单层架构的一个常见的变型就是网络数据库。在这种模型中，你可以通过某个网络直接定位数据库（或者它的某些部分），但是所有的处理依然是在本地实施的。

注意： 绝对不要试图将应用程序本身放在网络驱动器上。这在理论上是可能的——但是决不推荐这么做——因为这对网络造成了严重负担。应当将应用程序放在本地计算机上，然后使用链接表来访问网络数据。

一个网络数据库——它可能仅仅使用Jet数据库引擎，而不是SQL Server——允许多个用户同时访问数据。对于Jet数据库来说，同时访问的最大用户量是255个。在现实中，实际的最大用户量要依据他们所执行的操作，最根本地还要依据系统设计的效率。很显然，20个人以尽可能快的速度录入数据肯定要比50个人浏览销售和考虑产品类别带给系统的负担大得多。

减少网络负担对数据库模式的效率有直接影响。请记住所有的处理都是在本地机器上进行的；在某种意义上，你可以将数据库所在的计算机看作是一块远程硬盘。但是通过网络访问的响应时间肯定要比一个本地硬盘驱动器要慢很多。此外，网络本身的容量是有限的，并且系统上的所有用户必须竞争使用。因此你需要减少来回访问的信息量。同样，这在很大程度上是一个实现问题。如果你是直接参与实现的，那么我建议你查询在本书的参考书目录中列出的资源，以获得更多的信息。

但是，数据库模式的两个方面可以直接影响网络的性能：数据库对象的位置以及恰当地使用索引。我们已经提到了将用户界面对象存储在本地的的重要性。除此之外，你可能需要考虑将那些不经常变化的数据拷贝到用户的机器上。

例如，产品列表相对比较稳定，并且对它们的引用也十分频繁。如果Products表并不是很大（几兆字节是可以接受的，但千兆字节就不行了），你可以考虑将它们拷贝到每一台客户机上。多数情况下，这将减少网络阻塞，提高性能。当然，你还需要提供必要的更新数据的机制，但是这不会有太多问题。如果系统是使用Microsoft Access实现的，那么复制操作就能够很好的完成任务了。

邮政编码、州名、国家以及企业的地区和分支，这些属性的列表对于本地存储来说也是很好的候选属性，因为这些列表都比较小而且稳定。它们同样也被频繁引用。（完全没有必要将仅由系统管理员使用的表，每年都在用户的工作站上拷贝一份。）而另一方面，销售订单、客户列表或者学生列表就不适合作为本地存储。这些表中的数据变化十分频繁，并且共享最新的数据是网络数据库的一个关键所在。

数据库模式中可以影响网络性能的第二个方面是使用合适的索引。你可以将索引当作一种袖珍表，它是以某种特定的顺序维护的。索引仅包含需要定制记录的字段以及一个指向实际表格记录的“指针”，如图13-3所示。

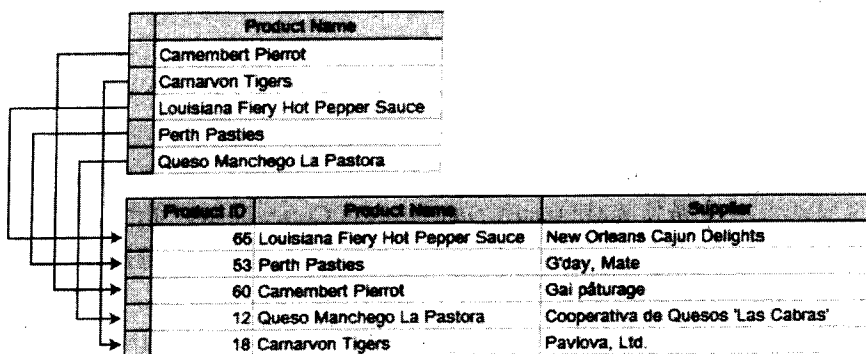


图13-3 一个索引就是一种袖珍表，包含以特定顺序存储的记录

请注意，我在这里使用“指针”这个词是比较宽泛的。这个指针和“内存指针”或者“对象指针”等等用在编程中的术语是不一样的。事实上，索引的物理实现和这里描述模型

并不十分切合，但是这个模型也起到一定作用。如果你真想知道里面的细节，那么《Microsoft Jet Database Engine Programmer's Guide》是一本很好的入门书籍。

如果不是物理地重组基本表中的记录，在大多数情况下某个任务将会相当耗时间，Microsoft Jet仅仅排序索引文件。这样做非常快捷并且也允许快速而简便地访问基本表，因为在任何给定的表中都有不止一个索引。

SQL Server提供了一种特殊的索引，称作**聚簇索引**，它可以控制数据的物理排序。每一张表仅有一个聚簇索引。

从性能的角度来说，索引的重要性在于大多数数据库引擎仅使用一个索引就能执行操作，而不需要读基本表本身。这甚至对于单机系统（哪怕从本地驱动器中读取数据会耗费一些时间）都能够显著地提高性能，并且在网络环境下，这种提高可能就是关键了。

举个例子来说吧，比如你有一张Customer表，包含100 000条记录，每一条记录1500字节长。应用程序需要查询一条特定的记录，比如是Jones Construction，他的CustomerID是JONSCON，那么你可以执行如下语句：

```
SELECT * FROM Customers WHERE CustomerID = "JONSCON"
```

如果该表没有索引或者没有声明主码，那么Jet数据库引擎必须逐一读取这100 000条记录来确定哪一条符合这个特定的条件。因此至少就会有150兆字节的数据会通过网络传送。如果该CustomerID是有索引的，就能直接添加或是声明它为主码，Jet数据库引擎就只需要读取索引，这可能只有几千字节，并且能迅速在基本表中定位正确的记录。

通过使用索引可以大大提高性能，但是也不能滥用索引。使用索引会存在一个额外的负担：每次添加或者更新一条记录的时候，数据库引擎就必须更新表中的索引。通常这种负担是可以忽略的，但是如果在任何一个表中都使用了过多的索引，那么它可能就开始影响性能了。极端一点来说，维护索引所需要的时间将会超过使用它们所节省的时间。

13.1.2.2 两层架构

在两层架构中，数据库和数据库引擎都位于远程计算机上。它们可以位于相同或者不同的计算机中。事实上，数据库可以分布在多台物理计算机上；但逻辑上，该系统依然是两层的。这种架构只有使用SQL Server或者另外一种数据库服务器比如Oracle才可能实现，使用Microsoft Jet是不行的。

乍看起来，网络数据库和两层系统之间的差异并不大。在远程计算机上使用Microsoft Jet或者SQL Server的最大问题是什么？其关键之处在于网络数据库中，所有的处理都是在本地工作站上进行的，但是在两层系统中，处理过程是分布在两个处理器中的。工作站负责处理用户交互，而远程计算机处理数据访问。SQL Server执行所有的数据操作，包括执行查询，并且将结果返回给客户工作站。鉴于这个原因，两层数据库系统可以更好地被称为**客户/服务器系统**。

为了更清楚地区分网络数据库和两层系统，以以下SQL 语句为例，这在我们讨论索引的时候用过：

```
SELECT * FROM Customers WHERE CustomerID = "JONSCON"
```

在一个网络数据库中，Jet数据库引擎将会读取索引（假定已经设有索引），然后确定正确的记录，并取回记录。索引和记录都将通过网络传送。在客户/服务器系统中，应用程序将该

语句提交给SQL Server并且得到返回的正确记录。仅仅只有记录是通过网络传送的。(当然,实际发生的情况一定会更复杂些,但是这个简单的模型也能够说明问题。)

针对像这样的查询,这两种架构的任意一种所表现的性能都十分出色,以致你可能看不出网络数据库和客户/服务器数据库之间的太多差别。但是如果是复杂的查询并且是多用户的操作,使用客户/服务器系统将会明显提高性能和响应速度。

除了可以减小网络阻塞之外,响应速度在客户/服务器系统中也能得到很大提高。因为当服务器忙于计算一个请求的结果时,工作站可以做其他的事情,比如响应其他用户的请求。反之亦然,当工作站忙于响应用户的时候(或者等待用户执行什么操作的时候),数据库服务器就可以自由地处理其他请求。SQL Server在很多方面比Microsoft Jet都要复杂的多,但是这样的系统响应效率正是客户/服务器系统比网络数据库能支持更多的用户的原因所在。这也是整体强于部分总合的另一个实例。

为了让客户/服务器系统发挥作用,必须尽可能多的将处理放在服务器端。然而,在网络数据库中,将查询存储在本地会更有意义一些,在两层客户/服务器系统中它们应当保留在服务器端。

如果你通过Access连接到SQL Server,那么你需要十分注意那些将在本地执行的请求的发布。例如,一个包含用户自定义函数的SELECT语句最好是在Access中执行,而不要传送到SQL Server中,因为SQL Server在SELECT语句中并不支持Access函数(Microsoft Jet同样也不支持,这就是为什么一个包含用户自定义函数的查询不能通过Visual Basic执行,即使该查询存储为.mdb文件)。如果你强制进行本地执行,那么将失去所有数据库引擎实施数据操作的优势。

从实现的观点来看,在构建客户/服务器数据库中还存在很多需要考虑的问题。同样,我推荐你参考关于这个主题的众多优秀书籍中的某一本,它们其中的一些已经列在本书的参考书目中了。

13.1.2.3 多层架构

在两层系统中,如果正确的将处理负担分散在两个系统中,那么它可以显著提高应用程序的性能和响应效率。把负担分散到其他的系统中能够得到类似的益处。再看一下图13-1所示的四层代码架构,通常会把事务界面层和外部访问界面层的组件分布在附加的中间系统中。

不幸的是,实现的复杂度似乎也是呈指数上升的。连通性、安全性、过程管理——所有这些问题在你转向三层或者更多逻辑层的时候会变得无限复杂。由于这些系统的复杂性通常需要额外的不同类型的服务器,比如Microsoft Transaction Server,因此这些系统通常被认为是“多层”。(显然,有多个物理层,就好比有多个丈夫一样,超过三个就不再数了,统称为“多个”。)

幸运的是,这种实现的复杂性仅仅是一个实现问题。你的开发团队可能决定在篮子编织上探索一个新行业,但是多层架构并不会对数据库的设计产生很大影响。你必须特别严格地保证代码架构的逻辑层次的清晰,但是一个在两层环境中运作正确的数据库模式不需要改动就能扩展到多层架构。

13.1.2.4 分离的架构

通过Internet或者某个局域网部署一个数据库基本上是多层架构的一种特殊形式。但是具体技术是不同的——你将使用HTTP作为传输协议,并且用户界面更像是Internet Explorer而不是Access——但是这两者的架构在逻辑上是十分相似的。

部署在Internet上的数据库和在较传统环境下的数据库之间的最大差别在于Internet是无国界的。与一个Internet应用程序不同，在典型的客户/服务器环境中，应用程序在最开始的时候将要求一个用户名和密码，之后利用这些信息连接到SQL Server。一旦建立了该连接（假定用户名和密码已被接受），服务器通常会在整个会话阶段保持连接状态。只有维持该连接，服务器才知道该客户是谁，并且当客户发出请求的时候，它可以恰当地做出响应。这种“我知道你是谁”的业务被称作是一种状态并且它由数据库服务器维护。

但是，当数据库系统被部署在Internet上时，数据库服务器将不再维护这些状态信息。每当应用程序对服务器发出请求的时候，它必须重建连接并且重新标识自己。一旦数据库服务器已经处理完该请求，它将清除所有做出请求的应用程序的信息。

分离的架构过去常被限制用在Internet和局域网上，但是这是通过ADO.Net强制执行的一种架构，它不管物理架构是怎样的，因此它们是很难避免的。

在大多数情况下，每次请求进行的一次新的连接所造成的负担对数据库系统的影响甚微，并且它也不会影响到数据库模式。但是Internet无国界的特性导致另一重意义，使得它不仅对应用程序有影响，并且还可能需要改变数据库模式。

大多数的Internet应用程序的目标都是一个**瘦客户端**。这意味着应用程序在客户端应当尽可能地少做处理，通常只处理用户界面。但是考虑到返回大量记录的情况，它们也许不能合理地在一屏上显示出来。在传统的应用程序中，结果集是缓存的，要么在客户端要么在服务器端。

但是在Internet应用程序中，这些结果不能在服务器端进行缓存，这是由于服务器不知道将下一批数据送往哪里。如果它们在客户端缓存，那么数据处理组件（事务界面层和外部访问界面层）必须也位于客户端，而这些组件肯定不会太“瘦”。事实上，它们是相当臃肿的。

ActiveX Data Object (ADO) 以及Net Framework都提供了一种称作**页面调度**的机制来处理这些情况。页面调度允许你返回结果集中特定数目的记录。这很像一个标准SQL SELECT语句中的TOP N子句，除非你还有一种叫做“middle N”的功能。

在服务器端，当每次重复执行查询时，用户都会请求一个新的页面，这就会导致页面调度。对于那些有较快的响应时间的查询，这就不存在问题。但对于那些复杂的，计算起来相对较慢的查询来说，你就有大麻烦了。一个应用程序让用户等若干分钟才响应尚可接受。但如果让几千位用户在每次他们想要查看下五条记录的时候都需要等上好几分钟，那么这个程序就应当属于垃圾产品了。

如果你在一个分离的应用中面临一个复杂查询，你有好几种选择。首先，绝大多数情况倾向于在该查询的短暂生命周期内优化它。比如创建临时表，反规范化数据等，尽可能将响应时间降低到一个可以接受的水平。

如果这样行不通，那么你就别无他法，只有创建一个**胖客户端**，将数据处理组件转移到客户端。这种架构允许你在客户端缓存查询结果，重新产生一个类似网络数据库的环境。

但是，作为一个一般原则，胖客户端更适合于一个局域网上而不是Internet，尽管有Microsoft的Web Services Architecture。很多人都很厌恶下载代码组件，但是这种情况不太可能在一个公用的应用程序中出现。如果它确实出现了，那么你必须期望你的内容的价值足够超越它们所带来的阻碍。

13.2 数据库模式组件

一旦完成了概念数据模型并且确定了系统架构，那么就有了构建数据库模式的绝大多数信息。数据库模式是对将要包含在数据库中的对象的一种描述。如果你已经选择了单机数据架构之外的其他架构，那么数据库模式也将定义每一个对象将要部署的位置。

如果你是使用Access来实现系统，那么你的数据库模式将包含每一张表、查询和联系的定义。它不会包括系统的窗体、报表和代码组件的描述，即使这些也存在在.mdb文件中。如果你使用SQL Server实现系统，你的数据库模式将包含数据库中的每张表、视图、存储过程以及触发器。

13.2.1 定义表和联系

在数据库模式中表的定义是直接 from 概念数据模型中派生出来的。实体将变成表，并且表的每个字段就是实体的属性。对于大多数情况来说，这个过程就是简单的直译。唯一需要特别注意的是约束、联系和索引。

13.2.1.1 约束

作为概念数据模型的一部分，需要为实体、属性和域定义约束。正如我们已经看到的，是否在数据库模式中实现这些约束依据的是系统架构的形式。如我所说，一些设计者倾向于只在四层模型中的数据界面层和事务界面层，或者三层模型中的业务服务层实现所有的约束。

在大多数环境中，建议你在两端实现这些约束。假定你同意我的说法并且已经决定在数据库本身上包含约束，那么就将它们作为数据库模式的一部分来定义。我们在第4章详细讨论了如何实现数据完整性，但是在这里仍旧需要重温一下。

大多数域和属性约束将成为数据库模式中字段级的约束，通常在Access中作为有效性规则。如果你选择使用SQL 语句，而不是DAO或者Access用户界面来创建数据库的话，Access也支持由SQL Server使用的CHECK约束子句。

实体级的约束通常成为表约束，同样地，或是作为有效性规则或是通过SQL CHECK约束来实现。你可以通过为每一张表定义一个主码来实现实体完整性约束，它指定实体的每个实例都是唯一确定的。

不论你是使用SQL Server还是Jet数据库引擎来实现数据库，都可能发现某些约束是无法作为表定义的一部分在概念数据库模型中定义的。在SQL Server中，可以使用一个触发器来强制实行这个约束。由于Jet数据库引擎不支持触发器，因此需要将这些约束作为应用程序的一部分来实现。

13.2.1.2 联系

我们在第3章已经讨论了在关系数据库中对实体间联系建模的方法，在本章还要讨论它。第一步永远是在外部关系中包含来自主关系的一个唯一标识符。在数据库模式这一级，这意味着在外码表中包含主码表的主码字段。

一些设计者不愿这么做，他们倾向于只在应用程序中处理参照完整性，而不是让数据库引擎去做。和所有的数据库有效性检查一样，在自己的工作中这两种方式我都采用：在应用程序中检查有效性是为了可用性；而在数据库引擎中检查有效性是为了安全性。我想如果我是一位男士，我会同时穿上皮带和背带的。

我们之前讨论了索引对于系统性能的重要性。每一张表应当至少有一个索引，在声明主码的时候数据库引擎就自动创建了索引。除此之外，还应当为用于连接表的字段或者字段组合创建索引。对于描述主关系的表，这通常不是问题，因为用于连接表的字段就是主码。但是，你可能需要在描述外部关系的表中声明额外的主码，因为在外部关系中用于连接的字段或字段组不一定组成整个主码。

如果外码字段或者字段组参与了主码但并不全是主码，我会在外码上定义一个单独的索引。例如一个OrderItems表通常有一个主码：{OrderID, ItemID}。即使在大部分情况下，主码索引可能被用来连接主Orders表（当然，我认为所有情况都是），为了确保可靠，我仍然会在OrderID列上创建一个单独的索引。

任何用来对数据进行排序的字段也应当建立索引。例如，客户列表通常会按客户名称排序，而订单会按日期来排序，即使这些字段无一参与主码也不作为连接的某部分，为这些字段建立索引也会使得排序过程更简单和有效。

在创建索引时走向极端也是可能的，所以在这里要十分小心。请记住虽然在维护每个索引所带来的负担很小，但却是积少成多的。任何将经常用于对表进行排序的字段都应当创建索引，但是你也可以使用SQL ORDER BY子句来排序记录，而不使用索引。

实际上，每张表中的最大索引数目依据的是表更新的频率。（只有当一条记录添加进来或者对索引字段进行更新时才会造成负担。）对于一张类似Orders的表，系统更新得较为频繁，我会十分仔细地维护不少于10或15个索引，包括那些用户支持连接的字段以及主码。另一方面，你可能需之在Product表中使用更多的索引，它更新的并不频繁，但是在整个系统中很多地方都会用到它。总之，你必须依据数据使用的情况而定。

13.2.2 视图和查询

Access和SQL Serve都提供了一种机制来存储SQL SELECT语句。这些存储的语句在SQL Server中称为视图，在Access中称为查询。（我在这里称它们为查询，因为这是比较普遍的术语。）在多数情况下，使用一个已经存储的查询要比执行一条SELECT语句快得多，但情况并不总是这样，但那些不是的情况是很少见的，你可以将这一点作为一个一般规则。

开始可以通过为复杂的实体检查概念数据模型，来决定哪些查询要包含在数据库模式当中。记住，一个复杂的实体是一个单独的逻辑实体，为了获得高效率它需要用两张或更多的表来实现。你应当包含一个查询来反规范化模型中任何复杂的实体。它们大多数都是一对多联系中的表，比如Orders和OrderItems，但是你可能还有一些复杂实体，它们已经通过一对一联系分成子类了，因此还应当包含支持它们的查询。

用户总是很频繁地查询系统中主要实体的某些特殊记录——例如，一个特定的客户或者订单——因此，这是第二处需要将查询存储在数据库模式中的地方。所有这些一般的查找都应当由一个带参数的查询支持，它允许用户在运行的时候指定某个特定的记录。

有时候你需要为一个实体提供不止一个的“查找”查询。例如，用户可能需要通过OrderDate、CustomerID，或者OrderID来查找某个订单。这其中的每一个都应当通过单独的参数查询来支持。

另一方面，用户不能查遍所有的表。你应当在数据库模式中有这样一张表，它包含美国所有州的列表。这些查找表十分有用，而且用户会需要查找某个特定的州名的情况是不太可

能的。

你还应当在由应用程序实现的窗体和报表中发掘查询。你需要一些链接字段的查询，还需要那些类似用于支持在组合框中进行的查询。如果系统有依赖的窗体，那么还需要一个参数查询来支持它们。这种情况的一个例子可能是一个订单输入窗体调用一个对话框来显示客户细节信息。

基于系统的工作过程，你还想在你的数据库模式中包含实施某些动作的查询（也可能是SQL Server中的存储过程）。如果你知道系统将定期地将订单存档或者更新产品价格，那么通过查询或者存储过程比在运行中处理这些请求要有效的多。

额外的动作查询很可能在实现的时候被添加到数据库模式中。和索引不一样，一旦实现它们之后，查询和存储过程都不会带来负担，因此在添加到数据库模式中的时候，不要有任何顾虑。

请记住，系统开发并不是一个严格的线性过程。在实现过程中表本身的改变会造成一些问题（并且，开发过程越深入，问题就越严重），在模式中添加查询是很平常的，也是系统所期望的。

13.3 安全性

当理解了系统的工作过程并且构建了概念数据模型之后，必须考虑系统管理的需求。管理需求不会直接影响到数据库模式，但是它们仍然是在发布系统中不可或缺的业务规则。

在某种意义上，管理上的需求是“元需求”，它们关心的是系统本身，而不是系统建模的问题域。它们可以分为两类：安全需求，它决定谁能够进入系统；可用性需求，它决定类似系统必须在线的频率（比如一周七天每天24小时，或者是正常的工作时间），以及用户如何备份数据。由于可用性几乎完全是一个实现问题，因此我们在这里只讨论安全性。

一个安全模式的实现可能是一个十分复杂的事情。幸运的是，在Access Jet数据库引擎以及SQL Server都有现成的过程。更幸运的是，由于数据库引擎是与实现分开的，因此在这个阶段只需要考虑逻辑的安全部署，而逻辑层的原则是很简单的。

安全级别

首先需要确定安全需求的级别。注意我们在这里谈论的是数据的安全性，不是系统编码的安全性，后者是一个实现问题。处在最低级别的安全实际是一个毫无安全性的系统，它允许任何人在任何时候访问数据库。这显然很好实现和管理，因为你不需要做任何特别的事情。

但是，如果你的数据存在某些价值，那么实现一个毫无安全性的系统是十分鲁莽的。如果客户已经通过限制访问实现了一个网络安全模式，那么它可能还有些意义，因为没有必要重复安全防范了。

下一个层次是**共享级安全**。在这一层次中，可以给整个数据库指定一个密码，并且任何知道密码的用户都能完全的访问系统。这也很好实现和管理，只要周期性地改变密码就可以了。共享级安全在很多情况下已经足够了。

用户级安全对整个数据库提供了最离散的控制，虽然它需要花更多的功夫来实现和管理。用户级安全允许系统管理员在每个对象上为每位用户指定特定的权限：“Joe可以在Customer实体中添加和编辑信息，但是却只能浏览Orders信息。Mary可以在Customer和Orders实体中

添加和编辑信息。不过Joe和Mary都不能删除任何类型的记录。”

事实上，称此为“用户级”的安全性在某种程度上有所误导。安全权限可以分配给各个用户，但是它们还可以指定给个体所属的一般用户角色。这种机制对于实现安全性来说会更加有效，因为它不需要太多的管理成本。

使用该模型，首先需要定义用户的类型——系统管理员、订单输入人员、销售人员等等，然后确定对于系统的每个对象各个角色所具有的权限。没有必要为数据对象指定权限，事实上，这么做是不明智的。你可能决定销售人员需要添加、编辑和删除Customers表中的记录，但是你并不想使这些表本身发生混乱。你可以为它们创建一个Customer维护窗体，而不是表本身。这可以确保它们不会偶然绕过维护窗体提供的特殊处理。

通常允许人们浏览的只是数据的一部分。例如，你可能允许每个人查看Employee表的Name和Extension字段，但是只允许经理查看Salary字段。或者你会允许销售人员看到他们自己的客户所下的订单，但不能看到其他人的。为了适应这两种情况，可以对基本的数据表分配查询权限和拒绝访问的权限。

审核

除了控制谁能访问数据之外，可能还需要知道用户都作了些什么。这种需求可能就十分宽泛了。某些企业想要跟踪谁登录了系统以及什么时候登录的。其他某些企业需要一个详细的审核，记录哪些人做了哪些更改，另外还有一些企业需要两者之间的某些信息。

如何构建审核需求依赖于具体的要求。如果只需要简单地跟踪谁使用了系统，那么一个属性为UserName、LogOn以及LogOff的单一实体就足够了。只需要在用户登录的时候创建一条记录，并在他或者她离开的时候更新记录就可以了。

有时候，还需要知道谁添加了一条记录。这一点可以在主实体中实现，添加一个或两个额外的属性：CreatedBy或者可能是CreatedOn。

跟踪删除可能会复杂一些。在关系数据库中，可以有多种选择。你可以阻止用户删除实际记录而仅仅只做一个删除标志，也可以添加DeletedBy和DeletedOn属性。如果你想在将这些记录从数据库中删除之前先把它们复制到存档文件中的话，这项技术会十分有用。

另外，你可以允许删除，但是要在日志文件中写入必要的信息，如果你需要跟踪哪些用户登录了系统也可以采用同样的方式。当然，你可能不得不创建额外的属性。毕竟仅仅知道某人删除了记录是没有太大意义的，除非你有某种方法能够重新构建原来的记录。

如果你需要详细知道究竟做了哪些修改，那么需要使用我们在第8章讨论的跟踪对维度记录所做的修改的一种技术。

如果你打算使用一个Jet数据库来实现这些审核功能的任何一种，你就不得不禁止用户直接访问表，因为那样会允许他们绕过你的安全策略。SQL Server则没有这样的要求，因为它支持数据库触发器，这是不可能被覆盖的。

无论如何构建这些审核需求，你都必须考虑这些信息将被怎样使用、被谁使用，以及在怎样的环境下使用。显然，你需要限制对审核表的访问。你可能还需要在系统设计中添加工作过程来适合审核。系统管理员需要有撤销已作的修改的能力吗？使用报告是必须的吗？

以我的经验来看，大多数审核需求只是一张保险单，并且这些信息仅仅在意外环境下才会用到。如果是这种情况，你就完全没有必要扩展工作过程。系统管理员可以简单地交互式地使用Access，或者使用SQL Server企业管理器来手动审核数据并且实施任何需要的动作。

13.4 小结

在这一章，我们讨论了将概念数据模型转化为物理的数据库模式。开始我们介绍了给系统构建代码的两种架构：三层模型和四层模型，主要的着眼点是代码架构的选择对数据库模式会产生怎样的影响。

我们还探讨了几种可能的数据架构。一个单层系统会将应用程序和数据都置于同一个本地机上。一个单层应用程序可以作为一个独立的应用程序来运行，或者将数据放置在网络上，但只能被一个用户访问。使用Jet数据库引擎实现的网络应用程序逻辑上也是一层的，但是可以同时有多个用户访问数据。

可以使用SQL Server实现两层或者客户/服务器应用程序。在这种逻辑架构中，服务器实施数据操作，而客户端负责响应用户。两层应用程序的最基本的原则可以扩展到三台或者更多机器上，这就是所说的多层应用程序。

然后我们讨论了概念数据模型向数据库模式的转化。这通常是一个十分简单的过程，因为仅有的新信息就是将在数据库中实现的索引和查询的定义。最后，我们讨论了安全需求对数据库模式的影响，并且我们简单回顾了逻辑层安全模式的设计。

在下一章，我们将快速关注一下有关将系统设计与客户和开发团队进行交流的问题。

第14章 交流设计

除非你构建的系统只是你自己使用的，否则你需要将设计的结果和其他人进行交流。请注意我说的是“交流”，并不是说仅仅创建“文档”。当然，你将至少创建一份文档作为交流设计的一部分，但是我已经看到太多的分析者只是生成一份“系统设计文档”，其中包括数据字典、界面截图以及报表样式，但却不解释这些部分是如何结合在一起的。

在世界上所有的文档中，如果它们是晦涩难懂的，那么就不可能达成什么目标。你不可能通过阅读一本字典来学习一门语言，同样，你也不能通过阅读数据表来理解一个项目。

现在，我有一个观点：基本的书写交流技能对于这项任务是十分关键的，因此如果你对你的这种能力有任何怀疑的话，请找一本好书（我最钟爱的几本书已经列在参考书目里了）学习学习，或者在本地的某所社区大学报一个学习班。我保证花这些时间是十分值得的。如果你对英语语法的掌握是错误的，你的客户一定会怀疑你对其他事情的理解也同样很糟糕。（说教到此为止。）

14.1 读者和目标

理解你的读者对于任何写作都是重要的，对于交流系统设计而言就显得更加重要了，因为你很可能拥有若干不同种类的读者并且有不同的要求。

你需要考虑你和你的读者想要达成什么目的。你的客户主要想确定你已经理解了他们的需求，其次就是想要获得系统能达成他们的目标的保证。你的客户不需要（或者不想）理解系统如何实现细节。但是如果该文档是打算用来作为开发的基础，那么开发团队需要确切地知道那些客户十分关注的细节信息。

有时候，最好的解决方法就是准备多份文档：一份给客户，另一份不同的文档给开发团队。如果你使用的是一种迭代的开发模型，那么这种方法是十分好的，因为它与该模型很贴切。在大多数情况下，我通常写多份文档：

- 一份需求说明，主要针对客户，它以非技术的术语记录对系统的理解。
- 一份架构说明，客户和开发团队都要阅读，但主要针对后者，它详细阐明了组件之间的交互和依赖关系。
- 每个组件一份单独的技术说明，供开发团队使用。

对于较简单的系统，一份文档通常就足够了。只是一定要确保你已经考虑了每一位读者的需要并且提供了他们易于理解的信息格式。

14.2 文档结构

除非你是在一个有特殊文档标准的公司工作，使你不得不遵循这种标准，否则你的文档的最终结构依据的是它的范畴以及你的个人风格。它可以很简单，也可以复杂到你认为合适的程度；这是没有标准格式的。我在这一章中可以给你一些指导，但是我之所以这么做，是假定你会根据具体的需求来调整它们。

如果你已经根据系统分析,采用我所说的建议,那么你会发现这份文档很容易就分成若干定义好的部分(并不全是巧合),这些部分与之前第三部分中的章节正好吻合。出于管理的需要,你可能还想包含一个简介或者执行小结。

14.3 执行小结

在大型企业中,通常会有一个操控委员会来监督项目的开发。即便是小型项目,一般也会有一个管理职位的人来负责监督或预算,他并不直接参与项目的实现。如果有人在监督你的项目,那么直接针对这个人,在文档中包含一份执行小结是一个很好的方法。

这些人一般对系统的细节不太感兴趣。他们有一些特定想要询问的问题,并且你回答的越有效越好。管理者通常想要类似如下问题的答案:

- 预定系统解决的问题是什么?
- 这是最好并且是最有效的解决方案吗?
- 其他的方案是如何考虑的?
- 实现该系统要多长的时间?
- 它的成本是多少?
- 风险是什么?

如果你已经定义好了系统的目标和范围,回答第一个问题应该十分容易。你只需要在此重申它们就可以了。我喜欢尽可能清晰地列出所有可能需要包含在系统中而没有包含的东西。我发现稍后指明这些会更好些。

如果你是一名外来顾问,你可能不需要理会第二个和第三个有关其他解决方案的问题。但是,如果你的确涉及这些信息,那么包含一个考虑过的其他方案以及这个方案被拒绝的原因的概要描述是很有用的。管理部门认为对替代方案进行适当考虑会更可靠一些。

但是一定要简要地回答第二个和第三个问题。如果你已经对现有的应用程序进行了全面的评估,并且为了有利于一个自行开发的解决方案而否决了它们,那么,你很可能有关于成本、功能、支持等的详细比较信息。执行小结文档中是不适合放这些信息的,应当将它们放在一个附录里。事实上,整个执行小结不应当超过几页长。

回答关于时间和成本的问题可能比较困难,并且它通常很吓人。但是如果你考虑要管理者购置什么东西的话,回答这些问题就变得更可管理一些。如果只是开发一个简单的系统,那么在已经定义的范围的基础上回答这些问题是一个比较好的想法。如果是大型复杂的系统,你不知道需要多长时间以及多少成本,但这没关系。你只要知道下一步做什么,并且这就是在这个阶段你需要认可的所有东西。

如果你十分现实地说类似这样的话,“现在还不可能对整个系统估计实际的时间和成本,但是我们预计在x到y这个范围内。但是下一阶段只需要z就能完成,并且结果是……”,那么,这就够了。只要确保“结果是……”中的内容是切合实际的,并且对于企业来说是相当有价值的。以我的经验来看,人们对资助“额外的研究”会十分犹豫。

我还发现充分地解决最后一个有关风险的问题是建立信誉的最有效的方式,所以多花点时间考虑可能出错的事情吧。是否存在尚未解决的技术难题?事情会比预期花的时间长吗?多想想在哪里以及为什么。

现在回头看看现实。这些事情发生的可能性有多大?是的,有可能你的整个开发团队会

由于一场突发的淋巴结炎的瘟疫而失去能力，或者办公室可能会被一场突然的洪水摧毁。但是这些都是不太可能的。可能的是某些时候在项目进行过程中会出错的事情，并且管理者想要知道你已考虑了类似的问题而且有应急计划。你不需要把每件事情都列出来，可能只需用几段话说明最重要的两到三个问题，但处理风险问题是一件很重要的事情。

如果你是一名外来顾问，我建议你不要回避你胜任的任何问题。显然，任何考虑雇用你的人要问的第一个问题就是你是否能胜任你的工作。执行小结并不是谈论你资格的最佳位置，但是你应该说明非性能方面的风险。风险协商的细节全依赖于你自己，我不想给出任何建议。仅仅根据我的经验来看，直接说明这些问题会帮助你建立作为一名商业人士的信誉。

14.4 系统概貌

不管你是否已经包括一份正式的执行小结或者一份缺少结构的简介，文档的第一部分都应该是系统概貌。系统概貌是读者所需文档的类似部分之一。开发团队和客户都需要理解项目的整体范畴。不同之处仅仅是细节的某个地方。

如果你还没准备好一份执行小结，那么这部分的某些信息就应当被包含在系统概貌中。即便你已经准备了一份执行小结，你也可能想要进一步讨论某些问题。否则，一些问题——比如替代方案的比较或者风险管理——可能需要单独的文档来详细说明。

不论你是否包含了这些话题，在这部分你主要的目标就是建立系统的“总体形象”，之后你可能通过解释系统参数来完善它：比如系统的目标和范围，设计准则，还可能有一个工作过程的概述。

交流系统目标和范围并不是件难事。如果你已经理解它们了，则只要简单地用你的读者能够理解的术语写下它们就可以了。如果你还没有理解它们，你最好现在退回去查清楚。

同样，我建议将那些排除在系统范围之外的内容尽可能说清楚，包括那些可能之后需要实现的内容。任何考虑包含在系统中的事宜都应当列在这里，即使你已经决定将某个领域排除在系统范围之外了。一定要花些时间确认那些你认为可能已经考虑包含进来但尚未讨论清楚地内容。这正是保证你和你的客户在同一前提下操作的大好时机。

如果你已经准备了一份系统的成本-效益分析，那么它应当被包含在这份文档中，但没有必要在这一部分。这是一个风格的问题，但是我不喜欢在主文档中包含好几页的表格。如果只在主文档中包含总结信息——可能只有一到两个表格——而将其余的放在附录中，那么它会是一份可读性更好的文档。

对于目标和范围来说，也是一样的道理。你可能已经准备了一份功能的详细分析，被它支持的目标的交叉引用，并且根据目标的重要性进行了规范化。这是一个很好的工具并且在整个项目中都十分有用。但是如果该表格超过一页长，那么它应当属于附录而不应当在正文中。只需要包含一个文字描述的总结表格来告诉读者参考恰当的附录以获得更详细的信息。

14.5 工作过程

交流系统工作过程的最好方式依赖于你是如何获取它们的。如果你已经使用了一种大纲的格式，那么你可以在文本中包含它。如果你已经准备了工作过程图，那么你也要在文档中包含它们。但是一定要包含对你使用的符号的注解。在任一种情况下，一定要解释清楚“过程”、“任务”以及“活动”（或者任何你使用的术语）这些术语的含义。

不论你的工作过程的描述是何种形式，一定要包含对工作过程叙述性的描述。首先，准备一份叙述文档就是对正式描述的一次很好的双重检查。通过这种方式，通常你会十分惊奇地发现很多小错误。

其次，这对人们评估工作过程也是很重要的，依我的经验来看，大纲和图形适合粗略地浏览而不利于理解。图文并茂的形式会促进人们的理解，并且这两种形式有利于互相说明。

如果你打算对工作过程进行修改，请一定要同时包括当前的和新的版本，并且在叙述部分强调所有修改的内容。显然，你要解释为什么提出的这些修改可以提升工作流程或者解决问题。如果包含对你提出的修改的详细探讨，客户经常会指出在你的最初分析中所忽略的某些事宜。

工作过程的文档是你的不同读者的需求可能产生冲突的领域之一。开发团队期望以技术的术语来讨论：事务需要提交，数据项需要更新等。另一方面，客户希望这些过程是用他们所使用的术语描述的。偶尔这些可能会是相同的计算机术语，但是它们大多不太可能一样。

当产生疑问的时候，宁可认定客户是错误的。在实际情况下，让开发团队理解客户的术语要比反过来困难得多。如果你必须使用技术术语，那么请一定要在文档中充分定义它们。

不幸的是，对此说起来容易，做起来难。当你整天和计算机一起工作的时候，很容易忘记某些术语不是通常的用法。我的做法是用一个清单记录类似“事务”或者甚至“文件”之类的术语，并且在将文档提交给客户之前做一次最终检查。使用文字处理器的“查找”功能并不是件难事。存有疑虑的客户很少是满意的客户。

14.6 概念数据模型

在完成最初的系统分析之后，你可能会有一套实体联系（E/R）图、系统中使用的域的列表以及关于数据约束的一些记录。将这些内容组织在一个表述性的格式中是很容易的。实体分析就是一份充分的模型文档，但对于复杂的实体可能需要通过一个合适的E/R图来说明。我通常把域分析以词汇表的形式单独作为一部份，并在模型中适当的位置引用它。

这个地方不可避免的存在一页一页的表。并且这里存在一个事实，那就是虽然这是一个技术问题，但这也是你的客户无法避免的。而你所能期望的是使得这个过程尽可能的不那么费劲。

首先，尽可能少地使用技术术语，并且只在需要的时候使用。例如“表”、“字段”以及“记录”很可能是无法避免的，但是“实体”、“联系”以及“属性”最好避免。我知道很少术语是不精确的，但是它们通常太相近了。同样，要确保定义你使用的任何术语（当然最好不要给你的用户上一堂简要的数据库设计课程）。

实际上，这不是什么大问题。一旦你已经解释了每张表描述一样“事物”，并且这些字段就是有关那个事物的“细节信息”，那么你的客户一般就很好理解了。你可能遇到一个偶然的关于邮编的查询，它是作为字符串或者类似的某种类型来处理的，但是我倾向于在这些问题出现的时候以非正式的方式处理它。

如果这份文档有双重作用，它还是一份技术说明书，那么你还需要将开发团队需要的技术细节包含进来。我试图将这些细节与主表分离开，通常作为每个实体的一个子标题。客户不久就会意识到他们不需要理解这些东西。在这些情况下，我通常会告诉我的客户要检查属性列表的完整性和域分析的精确性，但他们可以忽略其余的内容。

理论上,客户还应当检查实体间的联系,但是实际上我很少让我的用户在这方面找出什么错误或者有什么误解,并且之后只有我和他们面对面地检查该模型时才让他们查看这些。这显得有一点不相关了。

如果能让用户来检查字段大小和类型的话,说明你的运气很好。但是即使如此,如果你足够谨慎,那么更安全的做法是安排与合适的人实施检查。

我个人认为这会十分乏味——“好了,你认为25个字符对于一个姓足够长吗?”,这之类的事情对于我来说很难说是个好时光——但是,保证这些事情的正确性是十分关键的,所以这是不可能避免的。

不过,你可以重点标识出你关注的事项。当人们查看50页的表格的时候,他们的注意力很容易分散,因此如果标注出你需要确认的项目,那就很可能获得更好的结果。

14.7 数据库模式

数据库模式中的信息对开发团队是十分关键的,但是由于它包含了极少的额外信息,因此它基本与客户无关。如果你不打算准备单独的文档,那么请考虑将表和查询说明放在一个附录中。但数据架构和安全说明需要由客户确认。

不过,维度数据库可以是“仅开发者”原则的一个特例。正如我们在第二部分看到的,维度数据库的数据模型依赖的是数据库的生成,并从中派生出他们的数据。客户可能很想去确认数据源,并且一定需要看到有关数据是如何派生出来的讨论,以及实现维度数据库对生成系统所产生的影响。

不论实现什么类型的系统,我一般使用图和叙述性描述的组合方式来写文档。这是两种交流形式互相加强的另一种情况。安全需求可以以一个简单的叙述性描述来存档,但是有时候如果安全结构很复杂的话,一个大纲或者一张图会十分有用。

14.8 用户界面

在你深入做用户界面设计之前,准备一份用户界面的草稿文档来与客户交流是个好的方式。关于界面设计,本书的剩余部分将详细讨论。但是,对于小型系统来说,一份草图不太可能延误设计过程。

即使你不愿实施一个正式的用户界面设计,虚拟一些标记为“草稿”或者“样本”的样板屏幕界面通常会十分有用。样板界面可以帮助用户看到预订的系统。但你需要十分谨慎。不论你如何强调这些屏幕截图仅仅是样板,它们在发布的系统中会发生变动,这些样板都会成为用户期望的样子。如果发布的系统看上去大不一样了,那么你所有的好意都可能会付诸东流。

仅当你提供的样板屏幕界面是之后从项目的范围中得到的,否则设定用户的期望通常是个难题。我让用户在最初的需求中就提出他们期望的界面,即使他们之后看到并且认定要排除某些界面。在这些情况中,我已经认识到需要在某个部分包含这些被去掉的界面,这部分可以被称作“不再包括的功能”或者类似这样的词。对于决定不包含的功能以及不包含的原因的一份简要说明是让每个人知道它们所在位置的额外保证。

一旦你已经定义了用户界面,就需要将它与用户交流了。实施的机制主要有两种:原型法和界面说明。我一般会同时使用这两种方法,如果没有其他原因,构建一个无功能的原型

对于准备说明文档是一种最简单的方式。

14.8.1 界面原型法

我发现对于一个新系统，交流界面的最好方法就是界面原型。很多用户，特别是那些没有多少计算机经验的用户，都很难从一系列的纸张上的界面就想到系统的样子以及它是如何运作的。给他们提供一个界面原型并不意味着他们就能做这样的飞跃。

原型可以有多种形式和大小，它们可以用在很多目的上。最简单的一种就是虚的屏幕界面以及菜单绑定在一起模拟发布系统的流程。在一个界面原型中唯一需要的代码是将这些界面连接在一起。所有的控件都就位，但它们无需与数据绑定，也不需要其他的功能。同样地，唯一的菜单命令就是显示一个对话框。（当然，这不一定正确。我通常还给其他的菜单二级窗口，上面标明“该命令将如何如何，这个功能在原型中还没有实现。”）

“无代码区域”的唯一例外是当物理显示是由数据决定的时候。例如，你已经设计了一个界面来输入和编辑客户详细信息，但是界面的细节需要依据该客户是一个个体还是一个公司。你可能决定这些控件的显示都依据一个用户在选项框中的选择。那么，你需要在该原型中实现这一点。

我通常使用我在最终系统中使用的前端工具来构建原型。因为我工作在若干不同的开发环境中，这种方式可以消除类似“哦，天哪，我忘了你不能在VB中实现多列组合框”的情况。

但是，使用开发环境是危险的。你在这里构建的是一个原型，而不是一个系统，因此可以采用任何捷径虽然它们在生成代码的时候可能是不可接受的。但是很难抗拒使用原型作为发布系统的基础的诱惑。

毕竟，所有这些界面和菜单都已经构建好了，因此不使用它们是不是浪费时间呢？错了。这些界面和菜单是原型，而不是构建的。如果你使用界面原型作为发布系统的基础，那么你将保留你采用的捷径承担风险，并且它们将来会转过头来麻烦你的。

因为这种危险的存在，一些设计者建议界面的设计应当使用画图工具来实现，而不是使用编程工具。例如，Visio对设计界面提供了一些有限的支持。

你应当使用任何最适合你的工具。对于我来说，我会选择我每天使用的编程工具。对于你来说，可能是一个绘图工具或甚至是一个描述工具比如Microsoft PowerPoint。（我有一位客户认真地对我的原型进行屏幕截图，然后将它们转化到PowerPoint中作为他的内部演讲材料。他没有在界面上添加任何东西，他就是喜欢PowerPoint。）重要的是你清楚你在做些什么——你在对设计做建档，而不是在设计系统。

14.8.2 界面说明书

虽然界面原型是一个很好的工具，提供给用户一个对系统运作方式的感性认识，但是它的功能十分有限。鉴于这个原因，它不能取代界面说明书。（不过，反过来是正确的。一份仔细的界面说明可以消除对原型的需求。）

和数据模型文档一样，界面说明书必须包括客户不可能完全避免的技术信息。对此，同样建议尽可能地保留最少的技术术语，并且将那些客户容易忽略的技术内容从文档的主体中分离开来。

如果你已经构建了界面原型，那么准备界面说明书就是件很简单的事情了。我在其中包

含每个界面的一个位图，一段对该界面用途以及它实现的任何处理的文字描述，还有一张表列出每个控件以及控件的数据源（如果有的话）。如果你还没有构建原型，那么你可以以其他某种方式来描述界面，但是其他的信息是一样的。

对于大多数系统，在文档中包含一个系统流程的概要是十分有用的。如果系统支持多个不同的工作过程，你可能需要为每一个过程提供一个界面的模型。通过标注工作过程图就很容易做到这一点了。

14.9 修订管理

你的设计文档在它成形前很可能要经历好几个版本。一旦它稳定下来，并且在你进入下一项目阶段之前，需要将该文档（或者文档集）置于修订控制之下。

注意“修订控制”和“冻结说明书”是不一样的，对于后者，我认为即使是小型系统也是不现实的。如果你预计了不可避免的修改，那么从长远来看你的工作会轻松很多。

实现修订控制的方法有多种。我一般拒绝编辑这些文档，我发现将那些对功能的修订部分作为其附件会更简便一些。如果修订是大范围的，那么重写并替换整个文档或者文档的某些部分会更合适，但是这种情况很少。

如果可以对主要说明书建立一个中心位置，那么在文档上进行修订和注释可能就有益处了。例如，你可以使用由文字处理器比如Microsoft Word提供的版本工具实现这一点。之后该文档就可以被放置在一个网络共享上或者在局域网内发布。

我发现将文档放在一个中心位置的唯一问题就是要保证人们都是对当前版本的文档进行操作，而不是在已经打印好的复印件上进行涂写。我承认，我常常因此很自责。

14.10 小结

本章，我对如何与客户和开发团队交流系统的设计给出了一些指导。你可能认为它们是某位设计者的一家之言。在这里描述的策略适用于我，但更适用于我讨论的任何其他方面，你应当使它们符合你自己的工作方式以及你客户的需要。

虽然这是第三部分的结束，但我们还没有结束我们对数据库设计过程的探讨。在第四部分，我们将转向应用程序中最关键的部分：用户界面。

第四部分 设计用户界面

第15章 作为中间媒介的用户界面

在你已经完成前面章节所描述的分析任务后，需要清楚地理解你所设计的系统都有哪些功能。在本书的第四部分，我们将关注那些在为系统构建用户界面时应当考虑的问题。

在本章开始部分，我们将介绍设计界面的一般方法和你需要考虑的不同模型。我在这里能够提供的信息毕竟有限的，为了提高你设计用户界面的技能，你需要借助一些其他资源。在本书的参考目录中列出了几本优秀的参考书籍，而你所在的当地的技术书店相信也有一些其他的参考书。

15.1 有效的界面

对于用户来说，系统的用户界面即是系统本身，任何其他的内容均是可以忽略的。因此，用户界面设计的好坏就成为衡量一个项目成败的关键。如果界面设计良好，你的用户可能会原谅系统实现中偶尔的瑕疵；如果界面设计失败，那么无论你的代码多么有效，都将无济于事。

在这里较为讽刺的是，即使你的确做到了这一点，也很可能几乎没有人会注意到，真正优秀的界面是不彰显的；而即使你没能做好这一点，也可能没有人注意到。很多计算机系统的界面，特别是数据库系统的界面都是很糟糕的，所以你的系统很可能也就是人们已经能预料的另一个普普通通的、稍微滥用的计算机系统。

那么你会有这样的疑问了，“既然没有人会注意到，那么为什么要如此费心呢？”，要知道，毕竟这是你的工作。这的确是冒着很大的风险，但如果你确实打算设计好一个计算机系统，难道不应该尽全力做好它吗？

设计有效的界面所需要的工作远远不只是简单地作一个数据库前端，有效的界面设计需要花费更多的功夫去实现。尽管这些不是必要的部分，而且它的成本巨大，并且它们并不全是物有所值的。

一个有效的用户界面将会大大缩短用户学习和使用该系统的时间。一旦系统实现了，如果用户不用太费力气就可以学会使用它，那么就会获得更高的生产效率。因此，一个好方法便是将以上这些事宜在项目目标中标示出来，因为它们的确会产生关键性的影响。

此外，那些贴切符合用户期望和工作过程的有效界面会大大缩减对额外文档的需求，通常这些文档是相当昂贵的。也许用户没有清楚地注意到你的用户界面有多完美，但他一定会意识到你的系统运行起来比那些拼凑的、设计糟糕的系统要好得多。这样，在下一个项目或晋升机会到来的时候，它就很可能对你有关键性的影响了。

那么，怎样才算是一个有效的界面呢？依我的观点来说，一个有效的界面能够帮助用户

成功完成他们的任务，并且没有其他任何情况的干扰。一个有效的界面不会将它的种种要求强加给用户，不会强迫用户必须按照它的特定规则来使用；一个有效的界面也不会要求用户仅仅为了使用它而学习一大堆无聊的东西；当然，一个有效的界面在运行过程中，不会出现意想不到的情况。

在本章中，我们将会介绍以上这三个原则，但是我们首先需要了解一下在考虑设计用户界面时一些十分有用的模型。

15.2 界面模型

Alan Cooper在他的一本极富创造性的关于用户界面的书《用户界面设计要素》(The Essentials of User Interface Design)中描述了用户看待系统（同时也是系统看待用户）的方式有三种模型：意识模型、直观模型和实现模型。在决定设计应用程序的用户界面时，这三种看待系统的不同方式都是十分有利的工具。

一个用户的**意识模型**描述的是该用户思考时发生了什么。当然，这常常不会十分符合实际的情况，但是没有关系。举个例子来说，我有个模糊的了解，我的身体通过“燃烧”食物来给我提供能源，就像汽车引擎“燃烧”汽油的方式一样。但我很清楚地知道它们的工作过程是大不相同的，不过我并不关心。我只需要将汽油加到我的汽车里来保持它运转，而将食物吃进肚子里以保证身体的活动。我的意识模型已经足够让我来处理这些事情了。

对于计算机系统来说也是一样的。不论我是使用打字机还是文字处理器，只要我敲击一个字符键，那么一个字母就出现了，这时候的意识模型都是相同的。当然，实际情况却是大相径庭。而实际发生了什么就是所谓的**实现模型**。所有在屏幕背后的事宜，例如杠杆的拉动或者代码的运行，都是实现的一部分。用户不需要关心，也不应当被强制去关心它。

而用户界面便是介于用户的意识模型和开发者的实现模型之间的**直观模型**。如果你愿意，可以称之为系统展现给用户的处理模型。设计用户界面的目的就是尽可能去掩盖实现模型的细节。它常常不太可能十分契合处理过程的意识模型，但是你设计得越贴近就越好。

如果你是由于对计算机十分感兴趣而阅读本书的话，那么你的意识模型就没有符合用户的意识模型。我在使用一个文字处理器时的意识模型是当我敲击一个字符键，一个统一字符编码的值就被存储到随机存储器（RAM）中的某个位置。这并不十分接近实现模型，但也和一般的文职人员想的不一样。

这在设计界面时是十分危险的：即使你不会直接涉及实现过程，你也一定会知道一些关于它的情况。你要么不得不寻找一个临时的巧妙办法来忘掉这些系统实现的细节，要么去找一个用户当实验对象，让他给你提供他的意识模型。

最好的方法是使用系统的原型来做一个形式上的可用性测试。虽然这几乎不可能，但任何形式的可用性研究都是值得的。如果你已经构建了一个原型，那么就找一些用户来使用它，然后询问他们使用的情况，你一定会得到一些意外的收获。如果你还没有构建原型，你可以要求实现一个较小的系统或者甚至使用一个纸上勾画的模型。但是我本人并没有通过使用这项技术取得太多的成功。我发现，当把交互式的屏幕和一份报告均展现在纸上时，用户常常在两者之间十分困惑。

当你已经尽可能的收集到足够的信息，并考虑在实现模型（实际情况）的什么地方与用户的意识模型产生冲突时，你就需要解决这些冲突。是否你使用了错误的术语？记得一定要

使用用户使用的词汇。是否在用户仅仅只想“改变地址”时强迫他们考虑“编辑记录”？这可能是使用术语的问题，也可能是系统结构的问题（我们将在下一章探讨这些问题）。

15.3 用户层次

我不相信会有人有意地去构建一个用户界面极不友好的系统，也许系统的可用性不高，但是不会有人愿意设计一个用户反感的系统。问题是“用户界面友好”只是其中一个好的方面，友好的表达方式并不能代表全部，因此，你还需要努力发现一些其他专门的定义。

关于“用户界面友好”的两个定义常被称为“易学”和“易用”。如果我们暂且将“易”的确切含义放置一边不顾，我们仍旧需要问问自己，“对哪些人来说是容易的？”。一个对于初学者来说十分容易学习的系统，对于一个专业用户来说并不一定容易使用。最好的方法就是考虑不同层次用户的需求，使得每个层次的用户对界面的不同方面均十分适应。

15.3.1 初学者

每个用户在一定程度上都可称之为初学者。极少数人是这种方式——他们或是从一个“新手”转向中级状态，或是他们会由于喜好其他人的系统而将你的系统全然抛弃。从这个意义上来说，你必须十分小心不要因构建支持初学者的系统而妨碍了更多的高级用户。

在初学者开始学习使用系统之前，他们需要知道你的系统是做什么的。最好的方法是将这些信息描述在主系统之外。对于简单系统，一个描述系统的介绍对话框就足够了。（只是需要确保你总能提供一个让该对话框永久消失的方法。）对于更复杂的系统，一个指导性的说明会更合适一些。

对于初学者来说，联机帮助不是一个好的选择。他们可能根本不知道它的存在，又或者即使他们知道，也不清楚如何使用它。然而，通过将联机帮助链接到介绍性的对话框中或者帮助菜单中，我个人倒有一些成功的经验。为了让初学者更方便，这些指导必须是面向任务的。初学者并不想知道Menu Item（菜单项）是什么意思，他们就想知道如何去创建一个简单的发票单。

15.3.2 中级用户

对于大多数系统来说，绝大部分的用户都属于中间级别的。中间级别的用户知道该系统是干什么的，但是他们时常忘了操作的细节。这群用户是你必须在用户界面中直接支持的。幸运的是，微软视窗界面标准提供了很多工具来帮助这些用户。

一个设计良好的菜单系统是提示中级用户系统能力的有利工具之一。快速浏览一下可用的菜单项就能立刻提示他们系统可用的功能，同时这也允许他们去初始化合适的任务。

其次，对于中级用户来说，联机帮助是另一个十分有力的支持。不幸的是，书写联机帮助是本书范畴之外的内容。但是在本书中，我将提到大多数中级用户将使用联机帮助的索引作为他们主要的访问机制。因此索引应该尽可能的完善。

15.3.3 专业用户

专业用户清楚地知道系统能做什么以及怎样做。他们主要关心的是如何使事情更快速地完成。在你的系统中，你设置的快捷操作越多，这类专业用户就越满意。依据我的经验，专

业用户倾向于面向键盘的操作，所以如果你想满足这类用户需求，那么确保在你的系统中提供一种只使用键盘就能使系统运转的方式。

此外，专业用户也十分喜欢自定义他们的工作环境。但是，提供这种功能是一项十分昂贵的开销，所以你需要在提供该功能前仔细衡量利润问题。如果你确实决定提供一定程度的自定义界面的功能，即便它仅仅是一种在屏幕上调整窗体的方式，那么请一定要确保在各种会话之间保持这种调整。最让人烦恼的事情就是在每次加载程序时都要重新调整所有的事情。

15.4 让用户管理

在“用户界面友好”的背后，有一种华而不实的术语表达方式就是“以用户为中心”。这又是什么意思呢？与“用户界面友好”不同的是，它有一个确切的含义，尽管也有一点含糊不清。一个系统如果是以用户为中心的，那么它总能对用户的请求做出响应，并且从不强制一种特殊的工作方式。

在此介绍上述原则可能还为时过早，下面用一个例子来说明这个问题。在我认识的人之中，一位我十分尊敬的开发者如此描述他确保用户可以便捷地为系统顺序输入数据的方法：除了第一个控件外，他将窗体上其他所有的控件均锁定；一旦数据输入到该控件时，才释放第二个，接着第三个，以此类推。

这种技术不仅与“以用户为中心”的目标相去甚远，同时它在长时间运行中并不起作用。事实上，我也很惊讶它在短时间的运行当中也不起作用。用户常常有足够的理由不按照特定的顺序来输入数据，这可能是当时数据不可用，又或者是在那时输入并不方便。

如果你强迫用户输入一些东西，那么他们也会照此输入。他们会输入一些系统将要接收的旧垃圾数据。因此，通过强制一个输入，你就已经冒犯了用户，强迫了他们，最终导致一事无成。我们将在讨论完数据的完整性后，在第19章重点关注这类问题的细节。

人为地加强数据的完整性是数据库系统从用户那里争取控制权的主要方式。第二种方式是一种强制的模型。模型是用来限制用户的交互的系统条件，在数据库系统中这样的经典模型就是添加、编辑和视图。当用户需要编辑他们刚刚查看过的记录时，系统却要求用户返回主菜单，这样的系统显然是十分没有效率的。而在编辑前要求用户去选择一个菜单项或者单击一个按钮同样也不合适。

不幸的是，很多设计者将这种形式作为教条一般来实行。避免用户进行异常的改变可能是一种错误的尝试，又或者是因为二十年前，使用添加、编辑和视图菜单项已经成为一种习惯，它们使得这种典范在Windows环境下成为不朽，然而它显然是不合适的。我强烈建议你认为用户很清楚他们在做什么。（我认为这是一个很让人吃惊的论调，但是请相信我。更为重要的是，请相信他们。）如果一个用户想要改变一条记录，那么就让他改好了。没有人需要在做他的工作前被强制去获得批准。

当然，如果你打算给予用户这种自由，你还必须给他们提供安全的环境。增加不同层次的取消操作是一种简单的实现方法。你可能甚至需要提供一个Revert to Last Saved（恢复到最后保存状态）的菜单选项，从而允许用户放弃所有对当前记录的改变。

同样的道理，我也不喜欢在保存记录前询问用户是否确认对该记录的修改，虽然在某些情况下它是合理的。整个“保存”这个概念对大多数用户来说都是陌生的。记住用户的意识模型，他们刚刚已经对记录作了修改，而现在你却询问他们是否想要做修改，这显然是十分

混乱的。

这种类型的确认信息是另一个不太完善的强制用户的例子。一旦用户明白了整个“保存”的概念，无论确认信息对话框在何处出现，大多数人每次仅仅就是习惯性地点击“确定”按钮，所以这样的对话框并没有太大作用。

在一些少数实例中，当我已经实施了确认信息（通常是在客户端的坚持下），我会提供一种机制来将它们关闭（通常是在属性对话框中设置）。

但是，一些大范围的修改是不能被取消的，或者至少不那么容易被取消。对一条记录的某个字段的意外修改可以很容易地修正，但对一个表中的所有记录的意外删除是不可能恢复的。如果你允许用户去执行一些你认为十分危险的操作，最好的选择是提供一种方法可以取消这些操作。如果这是不可行的，那么就一定要让用户确认这些操作。

你需要很明智地给“十分危险”下个定义，并且提供给你的用户足够有用的信息，以提示他们这些操作的内在含义。图15-1显示的这种消息对话框仅仅能给用户以警示。

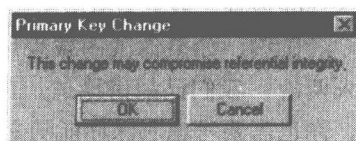


图15-1 这是一个毫无帮助的消息框

而图15-2显示的消息对话框就要好得多，虽然可能不是十分完美，但是相对要完善一些。这个对话框不仅使用用户的语言解释了当前的情况，并且除了“OK”和“Cancel”（一般用户将这两个选项分别俗称为“我是白痴”和“我是白痴，并致以我最谦逊的道歉”。）外，它还提供给用户其他的可选项。

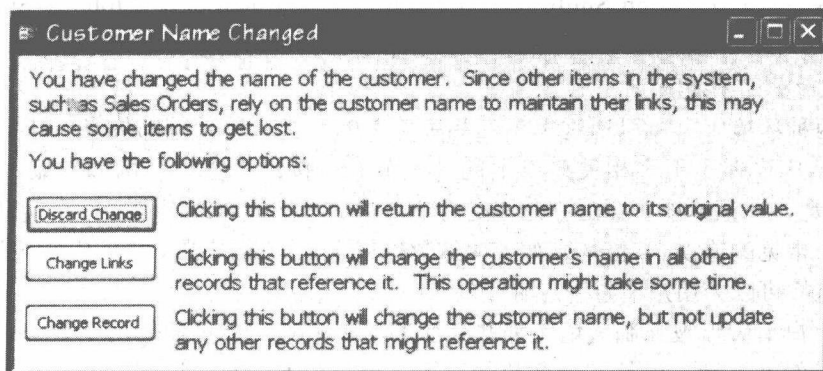


图15-2 该消息框解释了用户操作的内涵并提供了多个选项

15.5 减轻记忆的负担

人类特别不擅长记忆事务，这也是我们使用计算机的首要原因之一。由此可知，除非是绝对必要的，一个好的用户界面并不要求用户记住其他任何不必要的事项。很明显他们将不得不去学习（并记住）系统的功能，同时他们需要记住一些关于如何使用的事宜。但是在任何可能的情况下，你应当减少用户需要记忆的信息量。

一种方法就是依据Windows界面标准和规范。《软件设计的Windows界面准则》（The Windows Interface Guidelines for Software Design）提供了一套规范，你几乎没有理由去改变

这些规范，但也没有必要完全照它的方式做。Microsoft Office产品，特别是Microsoft Access，实际上采用的都是其他的标准。如果你采用不同于它的标准，那么请你确信有充足的理由。否则，你设计的界面很可能给你的用户提供不了任何帮助。

你可能会认为你的导航栏中的小工具会比Access中提供的播放器式按钮（Video Button，即用来移动记录的按钮）更酷。但是就要同情那些可怜的用户了。他们已经学会使用Play按钮来移至一条新的记录。而现在你却要使他们记住在你的程序里，他们必须点击那张难以置信的可爱图片，图片上是你那只怀有小猫的母猫。

当然，Windows界面准则也不过如此，它并没有包含所有的情形。但是在大多数情况下，你可以扩展一些通用准则来处理你遇到的特殊情况。当你的确要做出决定的时候，请确保遵循这样的规则。我们会在下一节讨论这种一致性的重要性。

很显然，如果你试图减少用户需要记忆的信息量，那么你一定不能让他们将同样的信息输入两次。我们已经讨论过，在这种情况下使用缺省值的方法。当用户必须完成一系列窗体时，你也需要考虑确保你能随之提供任何相关的信息。

此外，如果用户能够通过一个列表合理地选择信息，那么就没有必要要求用户输入信息。计算机是十分精准和考究的，用户不需要担心John Doe's Customer信息是否被记录为“John Doe”，“J.Doe”，或是“J.C.Doe”。但是需要注意我说的是“合理地”。在系统给一个组合框填入65,000条记录时，要求用户等候一段时间是不合理的。合理的做法是给用户提供一种过滤列表的方法，然后从一个易于管理的子集中选取记录。

当你将列表呈现给用户时，要尽可能包含足够的额外信息，特别是当列表项没有必要唯一时。用户不需要记住“John Smith”是那个居住在马德里的人，而“Johnny Smith”是住在米兰的。Microsoft Access的列表框和组合框允许你展示多列数据。在Visual Basic中，你还可以将相关联的字段连接起来。

在以上两种环境中，可以通过上下文相关菜单的方式来显示辅助信息。从技术的角度来说，这种显示应当在上下文相关菜单中通过一个命令按钮来实现，例如“显示明细数据”。（值得注意的是，在菜单项下面是不允许有省略号的，虽然它可以出现在一个对话框中。包含省略号是一个常见的错误。）但是，如果明细数据很少，那么就可以在上下文相关菜单中直接显示它们，这样可以为用户节省一点时间。

最后，用户不应当被强制去记忆那些晦涩的编码模式。你可能使用了系统生成的编码来保证每条记录的唯一性，但是在大多数情况下，用户根本没有意识到它的存在。除非这些编码有一些实际用途，比如是发票的编号，否则就不要显示它们。

很多机构和组织都开发了一套缩写代码来表达事务，例如产品的类别以及销售区域等。在系统中，我决定是否使用这些缩写的原则是它们是否会被使用在日常的会话中。你应当避免这样的情况发生：人们通常谈论“the Southwest region”，但是不得不输入“SWR”。仅仅当他们在会话中使用“SWR”时，在系统中才可以使用它。即使如此，我更倾向于在销售区域字段中同时允许“SWR”和“the Southwest region”两种输入方式。

15.6 保持一致性

在用户界面中的一致性不仅仅意味着在合适的位置获得File和Edit菜单，而且你的系统与用户交互的方式也应当保持一致性。当设计数据库系统时，你需要注意以下三个方面：第一，

在一张表中，用户是如何浏览记录的；第二，复杂的实体是如何表达的；第三，用户是如何初始化编辑和添加记录的。

大多数系统都是通过一系列的窗体来表达系统中的主要实体。例如，你可能有一个Customer窗体，一个Product窗体和一个Sales Orders窗体。你必须确定用户将如何浏览其中的记录集。除非你有充足的理由使用不同的机制，否则你应当在每个窗体中均选择使用同一种方法。

在Access和Visual Basic中，绑定窗体默认地都是显示所操作记录集的第一行记录，并且自带导航按钮以供用户浏览记录集。可能由于不同的原因，你想改变这种默认的界面方式。例如，也许你想仅显示一条记录，并提供其他的机制来选择要显示的记录——可能是一个单独的查询窗体或者一个组合框，用户可以选择一条记录。图15-3显示了使用后者的技术，它是摘自《Access开发者解决方案》（Access Developer Solutions）中的示例数据库。

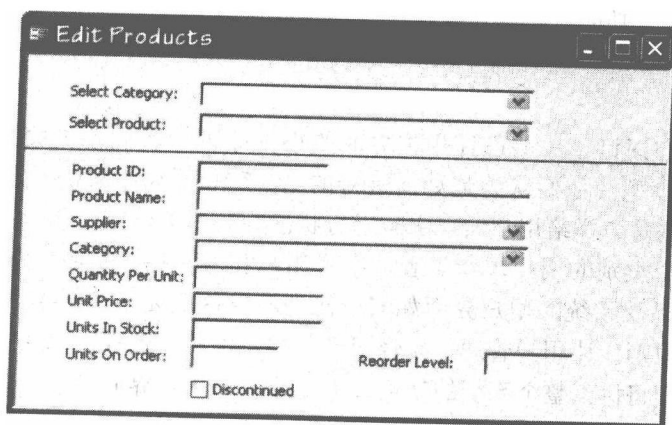


图15-3 有时一个组合框的确是让用户选择一个客户的很有用的机制，但必须以一致的方式使用它

改变默认的导航机制是不错的选择——事实上你常常会有很好的理由这么做。但是，一旦选择了某种机制，那么就应该在系统的所有窗体中均使用这种机制。完全没有必要在Customers窗体中使用标准的导航按钮，在销售订购窗体中使用Search Records按钮，而在Product窗体中用一个数据表单视图，这样只会让用户更加迷惑。

此规则的例外情况是：如果你有多种类型的窗体，例如，你可能决定让用来查询表格的窗体和用来输入主要实体的窗体有所不同。在这种情况下，只要同一类型中的界面保持一致，那么就不会给用户带来不适当的负担。

第二个需要保持一致性的方面是如何表达复杂的实体。如果在一个一对多联系中有使用多个表建模的实体（销售订购实体就是一个经典的例子），你应该以一致的形式将这些实体展示给用户。你不应当使用一个数据表单来表述销售订购记录项，而把客户的多个联系方式放在一个列表框中表达。

不幸的是，在这个方面的一致性更难以保持，特别是涉及超过两个记录集的时候更是如此。在一个单独的窗体上，有一个客户联系方式的数据表单，一个地址的表单，还有一个订购产品的表单，这样会使得窗体样式十分难看。在这些情形下，你就需要一些创造性了。你可以将每个数据表单包含在一个选项卡控件中，或者使用弹出式窗体来包含数据表单。你可能决定使用两种显示方法并且要可能地保持它们的一致性。

最后一个考虑一致性的方面是创建和编辑记录的机制，它在整个系统中必须确保一致。如果一个窗体的某个地方允许用户编辑，而另一个却要求用户通过点击编辑按钮或者选择一个菜单项来显式地进入编辑模式，那么，你着实带给用户带来不少困扰。当然，正如我们已经看到的，编辑模式通常也不是一个好方法。

有时候，一旦记录已被保存，的确有好理由锁定记录——就是说不允许在该处编辑。举个例子，你可能只允许用户在订购出货前修改订单，出货之后它们就成为了历史记录而不能再被改变。当用户想要编辑订单时，一些设计者就实施一种机制用来检查此订单是否已经出货。仅当所有的窗体都实施这样的机制时，这种做法才可以接受。

一个更好的解决方法是允许在该处进行编辑，但是当显示出货的订单时，要锁定该窗体上的所有字段。这需要更多的工作去实现，但是这些工作是由系统完成的，而不是由用户完成，因此是可以忽略不计的。并且，当那些锁定的历史记录不再应用到其他地方时，系统应当允许在该处实现编辑功能。

15.7 小结

在这一章，我们探讨了一些设计用户界面的根本原则。我们首先关注了三个界面模型：用户对系统的意识模型（他们认为是如何进行的），系统的实现模型（实际是如何进行的），以及直观模型（系统是怎样给用户展现实际运行状况的）。

接着介绍了三个类别的用户：初学者、中级用户和专业用户。我们主要讨论了每个类别的用户的特别要求以及系统的用户界面如何最佳地满足这些要求。最后，我们说明了用户界面设计的三个重要原则：让用户管理、减轻记忆负担和保持一致性。

在下一章，我们将讨论整个系统的架构，更多的关注用户界面设计中的实际问题。

第16章 用户界面架构

当你开始为你的系统设计用户界面的时候，第一个需要决定的是如何构建整个系统的界面——用户界面架构。在本章，我们将讨论几种较为标准的架构，这些架构在《软件设计的Windows界面指南》（Windows Interface Guidelines for Software Design）一书中有详细阐述，并且在很多流行软件中得到应用。

你可以开发你自己的用户界面架构，但是如通常所说的，当你考虑背离这些已存在的标准时，确信你有足够的理由这么做。记住一致性的要求，不但是在应用中，而且在各应用之间也要保持一致性，这样可使用户的使用更加轻松。

16.1 支持工作过程

决定如何构建系统界面的最重要的原则，就是你的选择必须建立在系统所支持的工作过程的基础之上，而不是建立在数据结构的基础之上。把注意力集中在你的用户要完成的任务上，并构建你的系统去支持这些活动。

很容易犯下这样的错误：你观察系统的实体-联系（E/R）图，看到其中有一个叫做Customers的实体，因此你就构建一个Customers窗体来创建和编辑客户记录。然后你构建了一个Order窗体，它对客户表有一个只读的引用。为了试图使系统“用户界面友好”，你允许用户从一个列表中选择客户名称，并且你依据客户表的信息自动为用户填写了剩余字段的内容。你的窗体可能就像如图16-1所示的形式，该图是来自Northwind 样板数据库。

Product	Unit Price	Quantity	Discount	Extended Price
Spegesild	\$12.00	2	25%	\$18.00
Chartreuse verte	\$18.00	21	25%	\$283.50
Rossle Sauerkraut	\$45.60	15	25%	\$513.00
			0%	

Subtotal: \$174.50
Freight: \$29.46
Total: \$203.96

图16-1 来自Northwind样板数据库的Order窗体

作为窗体而言，这是一个不错的形式，但是考虑一下用户的任务：他们需要输入一条销售订单信息。如果当他们打开Order窗体，发现系统还没能识别该用户，会是什么样的情形呢？他们不得不离开该窗体，去系统的其他某个地方输入这个新用户，然后重新打开Order窗

体，最后他们才能输入订购信息。如果Order窗体没有被关闭，他们可能需要记得按下Shift+F9（或者其他什么同样的按键）来刷新Customers列表。对用户来说这该多么痛苦！

一些设计者通过在组合框的“NotInList”事件中允许用户输入一个新名字来处理这个问题。当该事件被触发时，这些设计者便显示一个消息对话框来询问用户是否要添加一个新用户，并且如果用户确认要添加，则打开Customers窗体。

这种解决方案省去了几个击键动作，但是依然强迫用户暂停他们当前的工作（输入一个销售订单）而去做其他的事情（维护客户列表）。这就与让用户无干扰地完成手头工作的目标相去甚远了。如果用户输入一个列表中不存在的客户的名字，那些应当被填写的字段则会为空。（同时，这也足够提示该用户是一名新用户，而不必要显示消息对话框并发出警示声音而吓到用户。）在用户填完了相关字段后，系统应自动将新的客户记录添加到后台。

如果客户记录的所有字段信息均能从Order窗体中获取，那么你的工作就完成了。但如果还存在其他的额外字段（这是通常的情况），你可能决定询问用户是否愿意在填写完销售订购信息之后，再添加一些客户其他的信息。（但是要确保是在订购完成之后询问，中途打断是十分粗鲁的。）你应当在订购信息正在输入的情况下作出你的决定，换句话说，是在工作过程中。

如果销售职员正在录入订购信息，而新客户站在他面前，那么，当订购信息拿给客户之后就是获取那些额外信息的绝好时机。“由于您是一名新客户，您愿意提供一些您的相关信息吗？……”同样的方法，如果额外的客户信息（或者至少部分信息）在Order窗体中是用来数据输入的，那么当该窗体就在他们面前时允许用户很容易地输入信息，这样就免得之后从纸张数据中获取。

但是如果用户有一大堆销售订购信息需要录入，并且额外的客户信息不能立刻提供，那么大多数情况下他们会离开该对话框而不输入额外信息。不断提示用户去输入他们没有的信息是无用的，也是一种烦恼。一个更好的解决方案就是将这些未完成的记录作上标记，之后当信息可用的时候，允许用户很容易地找到它们。

你可能想询问用户当他们离开Order窗体后是否愿意直接转到Customers维护窗体（或者你系统中的什么其他叫法）。但是这么做的前提条件是，输入该信息是继输入销售订购信息后十分明智的一步。

16.2 文档架构

用户界面架构可以分为两种，其划分依据是应用程序的显示方式：一种是**单文档界面**（SDI）——应用程序在单个窗口中显示；另一种是**多文档界面**（MDI）——显示一个主窗口，其中能打开其他附加的窗体。

两种界面风格都各有千秋，不能说哪一种比另一种更好。在这一节中我们将讨论他们各自的优缺点。你选择的用户界面架构应当是基于系统支持的工作过程。

16.2.1 单文档界面

如你所想的一样，单文档界面展示给用户的仅是一个单一的主窗口，它可以通过显示额外的对话框来描述补充信息。单文档界面的方法适合那些倾向于维护单个逻辑实体的系统（这些实体可能又由任意多个数据库中的物理表格来描述）。例如，一个简单的用来维护雇员信息的系统最好是使用单文档界面来描述。

单文档界面有很多优势，因为一个单一的窗口对用户来说是很容易操作和记忆的。它遵循用户界面设计中以文档为中心的方法，该方法也是被微软的用户界面奇才们所推荐的。

单文档界面系统可以很容易地由Microsoft Visual Basic来构建，但它不能在Microsoft Access中直接实现，因为Access窗口中包含所有的窗体。然而，你可以通过在应用程序启动的时候最大化系统的主窗体并去掉最小化的按钮来达到单文档界面的效果。在Access 2000及其后续版本中，你还可以指定那些显示在任务栏中的窗体窗口。因此只要你愿意，通过仔细布置和管理，你的Access系统也能够成为一个单文档界面的应用程序。

16.2.1.1 工作簿应用程序

一个工作簿界面架构是一种特殊的单文档界面。在一个工作簿中，不同的数据视图在单一的窗口中通过不同的选项卡来显示，而不是分别在不同的窗口中。Microsoft Excel就是一个工作簿的很好的示例。

这种界面架构的优点是：它提供给用户一个安全环境，而不需要将用户限制在一个单一的窗体上。但在可接受的响应时间内，它能比较灵活地实现，但是抛开在实现过程中所提出的性能问题，工作簿的确是一个有用的表达一个对象的不同视图，或者表达相关对象集的不同视图（当用户不需要比较它们时）的有效机制。

例如，你可能想使用一个工作簿将夏季的月销售报表显示在一个选项卡上，在第二个选项卡中用一个饼图来显示不同产品类别的销售情况，而在第三个选项卡中用一个柱状图来显示到目前为止的年销售情况。这些都与一定的信息相关，并且它有理由期望用户愿意将报表作为一个组来查看。但是他们很可能没有必要同时查看所有的报表，因为不同的报表之间会有关联但并不是直接可比较的。

一个工作簿中的选项卡有一个隐含的次序，这是十分有用的。可以说，你所支持的一个工作过程肯定由许多离散的任务组成，进一步来说，这些任务通常（但不绝对）存在一个特定的次序。你可以在一个工作簿中为每一个任务设置一个选项卡。选项卡的次序与任务实施的次序一致，这样就不必强制用户去遵循哪个次序了。

另一方面，工作簿对于表达分散的个别工作过程不是一个好的方式——这些过程都有各自不同的活动，工作簿也不适合用来表达那些需要直接进行比较的信息。要注意的是，在任何给定的时间内，只有单一的一个选项卡是可见的，并且，你不要期望将不公平的负担强加给用户的短暂记忆之上。

16.2.1.2 Outlook风格的界面

还有一种比较特别的用户界面架构，自从我第一次见到Microsoft Outlook之后，我便称之为“Outlook风格的界面”。这种风格的界面将应用程序的窗口划分为两个窗格，一个包含图标集合，而另一个包含文档，如图16-2所示。

我喜欢应用程序的这种界面风格，它能支持多个工作过程。左边的图标栏给用户提供了一种快捷的环境，将各个栏划分成多个面板的方式，可以使得图标归属到不同的功能区。

不幸的是，Access和Visual Basic都没能提供固有的控件来实现这种风格的界面，尽管它在Microsoft Access数据库窗口中有所体现，如图16-3所示。不过，在这两个环境下，都可以通过第三方获取Outlook风格的控件。

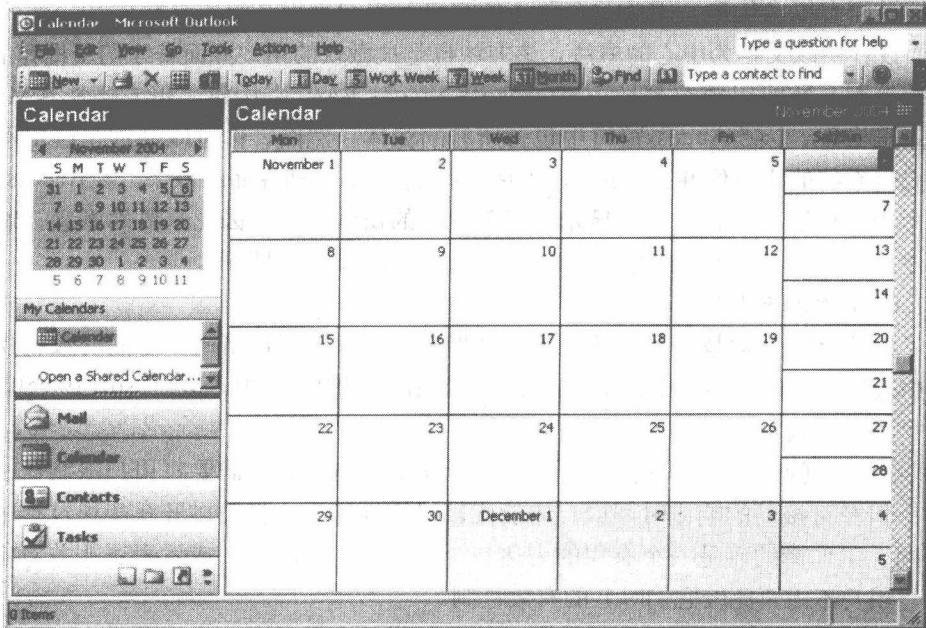


图16-2 Outlook风格的界面将应用程序的窗口划分成两个窗格

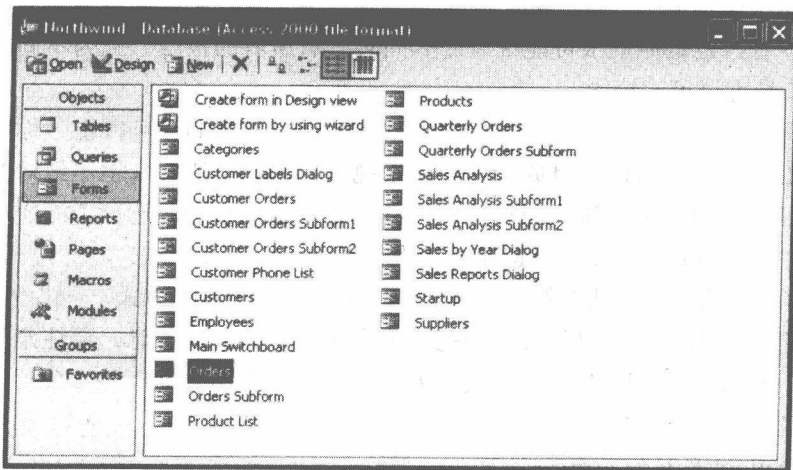


图16-3 Microsoft Access 数据库窗口使用Outlook风格的界面

如果采用这种架构，一个好的思路是允许用户隐藏图标栏，就像Outlook一样。图标栏是一种很好的导航机制，但是一旦用户加载了他们想要工作的文档，他们就倾向于在文档环境下多停留一段时间，但是这样图标栏就会占用已经十分有限的屏幕空间。

16.2.2 多文档界面

大多数的数据库系统都采用多文档界面的形式，它允许在一个主窗口中打开多个子窗口。多文档界面的子窗口可以包含多种不同的信息，例如一个窗体用来显示客户信息，另一个窗体用来显示订购信息。此外，它们还可以用来显示同一信息的不同视图，例如用一个窗体来

显示一位客户的信息，并用一个报表来显示给该客户的销售信息。最后，他们可以显示同一种信息的多个实例，例如用一个Customers窗体显示Jones的明细信息，而用另一个相同窗体的实例来显示Smith的明细信息。

如同单文档界面架构一样，构建多文档界面的应用程序也有多种方式，每一种方式都适合于不同的应用要求。我们在这里讨论几种主要的结构，但是要注意这些结构形式并不十分完备，因此，它们之间也没有必要互相排斥。

16.2.2.1 经典的多文档界面架构

经典的多文档界面应用程序的结构是在一个主窗口中包含多个相同或不同形式的子窗口。当用户需要比较多种不同的数据或者同种数据的不同形式时，多文档界面应用程序就十分有用了。但是多文档界面应用程序可能会使新用户产生疑惑：当用户在空白窗口中新建文档的时候会出现一些选择项，但用户从File菜单中选择“New”时却没有出现这些选择项。

Microsoft Word (Office 2000以上版本)的多文档界面形式允许一个命令行启动开关在应用程序启动时自动打开一个新的文档。如果你采用这种界面架构，那么你可能会考虑在你自己的应用程序中提供类似的功能。例如，你可以提供一个开关使Order窗体打开以便录入数据。

一定要提供一种将这项功能关闭的机制，因为它可能会激怒某些用户，对于他们来说这样的功能并不合适。此外，还要小心在实现过程中，不能随意地把错误提示信息显示给用户，否则他们会立即关闭窗口，更加糟糕的是在数据库中添加了空记录。

多文档界面的主要问题是在包含模型中的不一致性。父窗口显然可以包含在其中打开的子窗口，但是它代表的应用程序没有必要包含这些子窗口中表达的对象。在类似Microsoft Word这样的应用程序中，这就形成了一个更为尖锐的问题，因为在文件系统中文档都是绝不相同的对象。数据库应用程序通常在孤立用户方面要比复杂的文件系统做得好，但即使是简单的数据库应用程序也不能完全有效地达到如此要求。

举个例子来说，图16-4所示的是Microsoft Access中的Northwind样板数据库。假设一个用户一直在窗口之间前后切换，并且在它们之中做了一些尚未提交的修改。

如果该用户在File菜单中选择Save，那么会出现什么样的情况呢？只有在Suppliers窗口中所做的修改被提交，你知道这些是因为你知道菜单选项仅仅应用在当前窗口中。但是用户清楚这一点吗？当你告知“Northwind”要保存了，不论你在哪里所做的修改，“Northwind”都要为你保存吗？这种期望显然是不合情理的。

《软件设计的Windows界面指南》(虽然不幸该书已经过时，但仍旧是界面规范最好的定义书籍)书中详细阐述了将“Save All”选项添加到文件菜单中以保存所有打开窗口中尚未提交的修改。这无疑是一个解决方法，但是它至多是一个折衷方案。它仍然需要用户明白多文档界面应用程序和应用程序所操作的对象之间的差别。

这种差别是实现模型的一部分，而不是用户的意识模型，并且对于用户理解来说，这也是一个相当困难的概念。甚至一些思维比较复杂的用户也会对其中的存储内容感到迷惑。虽然我已经广泛地使用Microsoft Word很多年了，但我个人也仍然不清楚在该产品中，文档是以什么格式存储的，而图表又是以什么格式存储的。

然而，正如前文已经表述的，经典的多文档界面应用程序确实有其用武之地。它们依然是大多数需要同时打开多个窗口的应用程序的最佳解决方案。

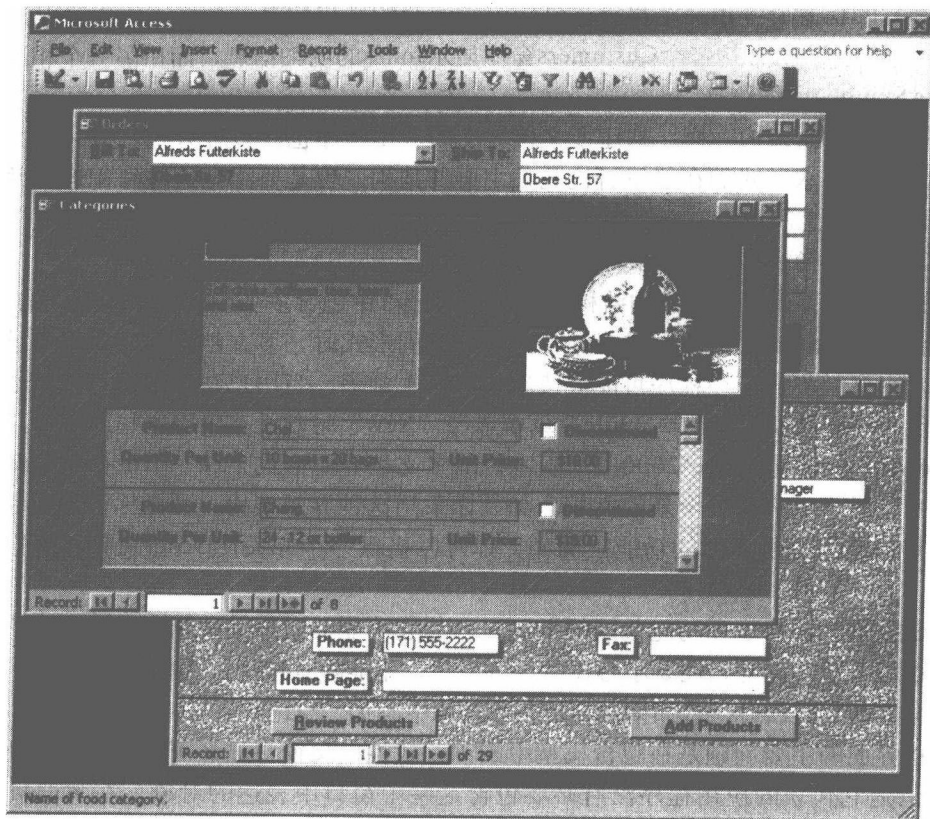


图16-4 多文档界面应用程序中的包含模型会迷惑用户

16.2.2.2 转向器界面

一个转向器应用程序是在程序启动时显示一个核心窗体，如图16-5所示，它同样是来自Northwind样板数据库。窗体上的绝大多数按钮是与其他窗体或报表相链接的。

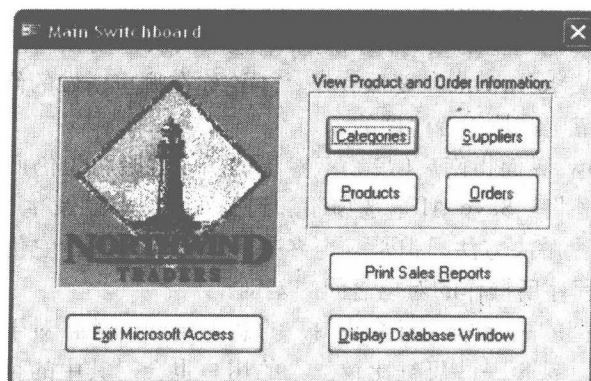


图16-5 在应用程序启动时显示的转向器

在Microsoft Access中创建新数据库时的New Database（新建数据库）向导就是使用的转向器结构，并且在Access环境中开发数据库应用程序时，这种结构的使用变得更为普遍。使

用Visual Basic也能十分方便地实现类似的功能。

我必须承认我对转向器的一种偏见，我有时认为它们像DOS一样十分沉闷。我也很担心他们会助长类似“查找记录/编辑记录/打印记录”一样的菜单结构，这对用户来说将是难以置信的单调乏味。

但是，转向器应用程序也是十分有用的，特别是当你的同一应用程序需要支持多种不同的工作过程，并且你不希望实现的复杂性像Outlook风格的界面一样，而你的应用程序又要求同时打开多个窗口时。在最高级别的转向器上包含链接每个工作过程的按钮是一个很好的机制，它能方便地引导用户到相应的应用程序。

和其他的界面架构一样，在设计转向器应用程序时需要记住的重要的事宜是：应当基于工作过程来构建转向器，而不是基于数据。你应当给用户想要实现的每项活动设置一个按钮，而不是为系统中的每个窗体或报表设置一个按钮。

16.2.2.3 项目界面

我认为一个项目界面就是一个不包含窗口的转向器应用程序。然而，一个“Project”窗口（当然，没必要这么称呼它）提供了一种在桌面上独立地打开窗口的机制。Visual Basic 6.0 SDI界面就是项目界面的一个很好的示例，如图16-6所示。

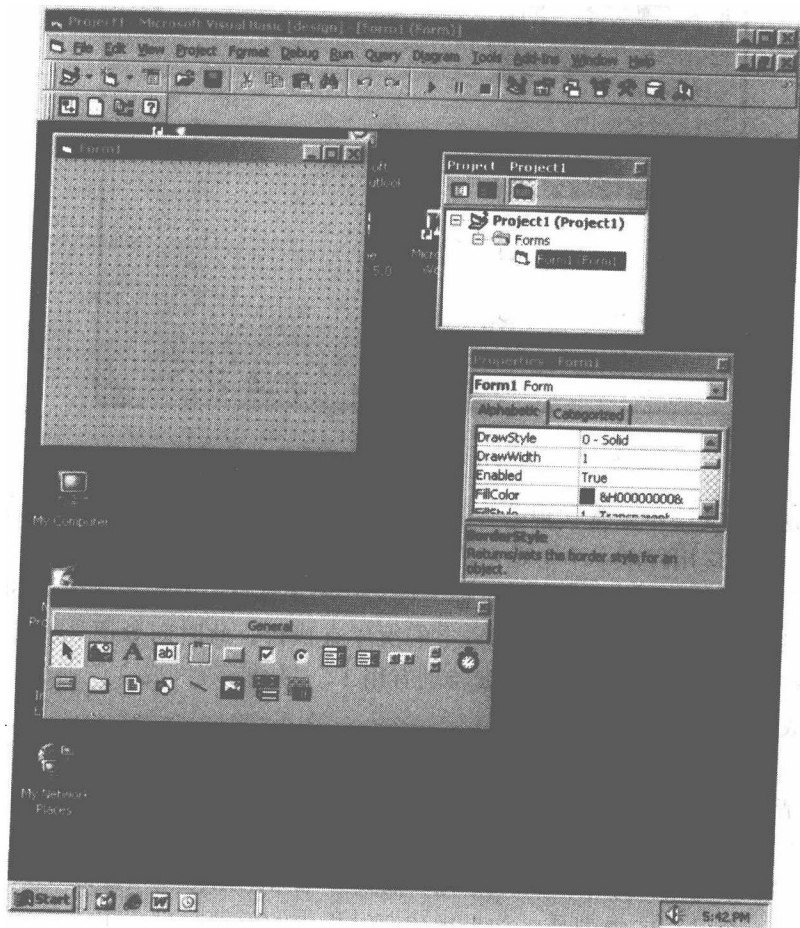


图16-6 SDI模型下的Visual Basic就是一个项目界面

一旦打开项目界面,这些独立的窗口就能显示在工具栏中,并且用户能够独立的管理它们。它们同样也受Project窗口的管理。当Project窗口最小化或关闭时,次级窗口也会受到影响。

项目界面消除了包含性,避免了经典多文档界面应用程序固有的模型不一致性,但是它带来了另一种不一致性: Project窗口和独立的次级窗口之间的一致。

考虑这种情形: 用户通过在Project窗口中双击来打开一个次级窗口,然后为了释放桌面空间而将Project窗口最小化。如此一来,用户刚刚打开的那个窗口也随之消失了。当然,它会一直存放在任务栏中,但是已使用户产生了迷惑。

如果你喜欢项目窗口的方式,我认为Access中的Database窗口模型要比传统的项目界面更胜一筹。该Database窗口提供了项目窗口的导航功能,既然它是经典多文档界面应用程序的一部分,那么它就不会有潜在的其他不良影响。

16.2.2.4 向导

最后一种界面形式就是向导,它在数据库应用程序中也占有一席之地。向导是由一系列显示在对话框中的页面组成的,如图16-7所示。

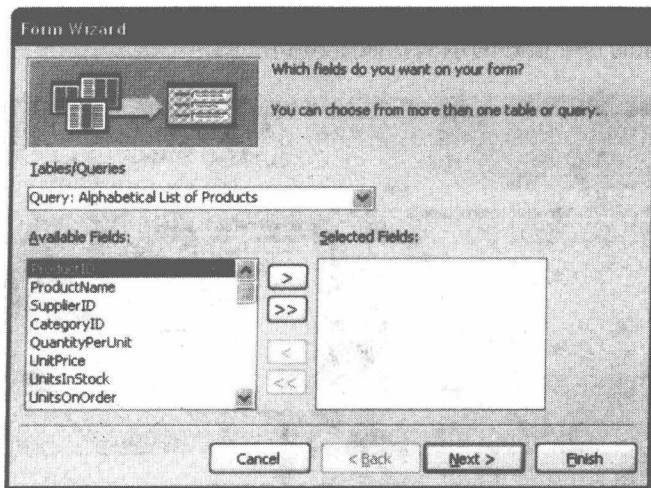


图16-7 Microsoft Access 2000 广泛地使用向导

向导时常被用来支持不经常实现的任务,例如安装过程或者硬件配置等。但是确实很适合用它来支持数据库应用程序中不常发生的工作过程。

向导也能用来支持较为复杂的工作过程。如果一个工作过程是由很多不同的任务组成的,而这些任务必须(或至少能够)以一个特定的次序来执行,此时向导便是界面的最佳选择。当工作过程含有多个条件任务时,向导就特别有用了:“如果条件a成立,则执行任务3和任务4;如果条件b成立,跳转执行任务6。”

仅当每个任务都能按照向导描述的次序来完成时,在复杂的工作过程中使用向导才是可行的。而这种情形要比想像的少得多。如果任务没有必要按照某种特定的次序进行,那么向导就不是最好的界面选择,或者至少不应当是完成任务的唯一界面。

当在一个数据库应用程序中使用向导时,需要仔细考虑用户在何时以及如何输入的数据是需要存储的。一个常用的模型是,如果用户在过程中的任何时刻点击了Cancel,那么系统就回退至向导初始化之前的状态——也就是说,任何用户已输入的数据均被丢弃。

根据不同的应用程序，你可以考虑在每个向导页面完成时存储数据，或者在用户取消时，给用户一个选择是否存储之前输入的数据的选项。我并不推荐以上任何一种方式作为一个通用规则，但是如果向导中有许多数据输入，这样做还是合理的。

另外，你也许会考虑允许用户临时停止向导而之后再继续。这是一个十分有用的解决方案，但是它的确增添了一点实现的复杂性，因为它要求一个临时的数据存储和一些决定用户在何处暂停的机制，还有从该暂停点重新开始的功能。

16.3 小结

本章我们探讨了在你的系统中构建窗体的不同方式。从最高层来说，可以有两种选择：单文档界面系统——在单一窗口中显示数据；多文档界面系统——使用多个窗口。

单文档界面系统有两种不同的形式：工作簿和Outlook风格的界面，它们每一种形式在某些特定情况下都有各自的优势。多文档界面应用程序中存在更多不同的形式，从经典的多文档界面到Access数据库向导应用程序支持的转向器架构，还包括以Visual Basic为例的项目架构。最后，向导对于支持不经常的任务以及复杂的工作过程也是一种十分有用的架构。

在下一章，我们的介绍将从窗体的组织形式转向窗体的结构本身，并探讨如何基于数据模型中的实体的结构来决定窗体的布局。

第17章 在窗体设计中描述实体

在前几章，我们一再强调注重工作过程，可能你会疑虑为什么一定要将那些数据进行建模呢？不必着急，在这一章，我们将关注如何使用数据模型来构建不同的窗体，而这些窗体的构建是基于实体在其中显示的方式。要根据工作过程来决定在何种给定的窗体上包含何种数据，但是一旦决定好之后，窗体的布局以及控件的选择将由窗体表达的实际数据结构来决定。

你需要做的第一个决定就是如何将一个窗体映射到实体联系（E/R）模型上。这个窗体描述的是单个实体，或是一对一联系中的两个实体，又或是一对多联系中的两个实体，还是更多的实体？这每一种实体架构都对应着一种窗体布局，关于这方面的内容，我们将在下一章讨论。当然，和我们在第16章中探讨的架构一样，这些布局仅仅是一些通用的准则，并不是硬性的规定。以我的经验来看，这些布局大多适合某些类型的数据结构的。

17.1 简单实体

在窗体上最容易被描述的是简单实体，它在数据库中只用一个表来描述。如果这样的实体参与到了任何联系中，那么，它要么在“多”的一端，要么在不包含在窗体中的联系的另一端。

举个例子来说，如图17-1所示的E/R图，它描述的是Customers实体及它的联系。

在应用程序的某处，你可能会使用一个窗体来维护客户信息。即使所有的客户数据都能通过其他的活动间接地获取，这么做也是十分合理和必要的。除了与Orders实体的联系之外，Customers实体在图中的所有联系中都处于“多”的一端，这样，在Customers窗体中就不应该包含Orders实体。Orders有逻辑的独特性，用户可以在系统的某个其他位置来维护它。

因此，处在所有实体的“多”端的Customers实体就被包含在客户维护窗体上。这就意味着其他某个表仅能包含一条记录，所以Customers实体可以被作为一个简单实体来对待。

这样只需要一个简单的窗体布局就可以了。你不必在客户维护窗体上添加子窗体或者表格控件，你只要简单地选择一个最能描述各个字段的控件（我们将在下一章关注这个问题），将它拖拽到窗体上，就算完成了。

当然，你不会仅仅只将它拖拽过来，还应当对所有的控件进行整齐地排列吧？比如，你会留有7个单位宽的页边距，每个控件之间有4个单位的间距，并且将最重要的项安置在左上角的位置。这下你明白了吧，没有什么技巧可言，除非你将窗体上的空间使用殆尽。

但是你不大可能将空间使用完。从技术角度来说，你可以在Microsoft Access窗体或报表

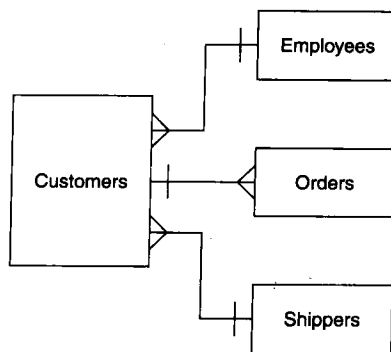


图17-1 显式与Customers有关的实体的E/R图

的整个生命周期内添加754个控件（已删除的控件也算在内），在Microsoft Visual Basic的一个窗体上最多可以添加254个控件（控件数组只算作一个控件）。

显然，你不会要求用户在同一时间内处理超过约25至30个控件或控件组。（注意我说的是“控件或控件组”，一个成组框（group box）可以包括三到四个单选按钮（radio button），而它仅作为一个控件。）那么，如果你的一个简单实体有75个属性，那么该如何处理呢？你必须在任意时刻只显示部分数据。

结构化描述有较多属性的实体的窗体的最简单方法，是将字段分解成不同的组或类别。我通常首先将那些绝对标识实体的属性挑选出来。这种方法中，我并不仅仅是指作为候选码的属性，还包括其他描述性的属性，这些属性能使用户确信他们在正确的实体上工作。

例如，在一个Customers实体中，这组标识性的属性可能包括姓名和详细地址，也可能是销售代表属性。对于一个Products实体，它可能包括产品类别、产品名称和产品描述属性。这组属性在窗体中应处于最上方，并且一直保持可见。

然后，剩下的那些相互关联的需要一同查看的属性就被分到一个组里。例如对于Customers实体来说，你可能将以下属性组合在一起描述销售方式：标准折扣和支付方式。这样就可以将它们分到一个组中，而将采购人员、销售经理等分到另一个组中作为联系信息。对于Products实体，你可能将技术规格说明组合在一个组，而把与包装相关的属性放在另一个组。

将属性组合合并进行分组后，就可以有多种选择了。可以在主窗体中为每一组属性设置一个单独的选项卡控件。这是一种我常使用的解决方案。选项卡控件给用户提供了大部分的环境——他们可以迅速得知附加信息是否可用以及这些信息的内容。但是，如果属性组超过了五个或六个，那么选项卡控件就合适了。

在这种情况下，你可能不得不将部分或全部的属性组移至次级窗体。次级窗体也是十分有用的，特别是当有些组包含过多的属性，而使得你无法在一个选项卡上合适的放置它们时。可以允许用户在主窗体上通过一个命令按钮来打开次级窗体，这种结构类似于一个转向器。但是，如果命令按钮过多，你的窗体可能再次变得不可用。这种情形下的一个更好的解决方案就是通过菜单来打开次级窗体。

当使用次级窗体时，一定要注意为用户维护当前环境。用户应当能十分确定当前显示的细节是属于哪一个实体的实例。早期用来标识实体的方式是在屏幕的次级窗体上不断的重复主窗体上的细节信息。通常没有必要重复用来标识实体的整个组——只重复那些能将次级窗体和主窗体衔接起来的信息即可。

另外，你可以制作一个次级窗体模型。这是系统中为数不多的几处使模型化较为合理情况之一，因为它可以帮助维护用户的环境。但是，模型化窗体通常会约束用户，这是你需要尽可能避免的。我不到万不得已是不使用这种技术的，比如当需要将为数不多的要求维护环境的属性包含进来，而确实没有足够的空间。

代替次级窗体的一种方式就是一个面板架构。这是一个相当简单的方式，它为用户提供了一种改变显示模式的机制，或是从一个菜单选择，或是更倾向于使用一个主窗体上的控件。图17-2显示了一个例子，它是Visual Studio中使用一个TreeView控件来控制显示的。

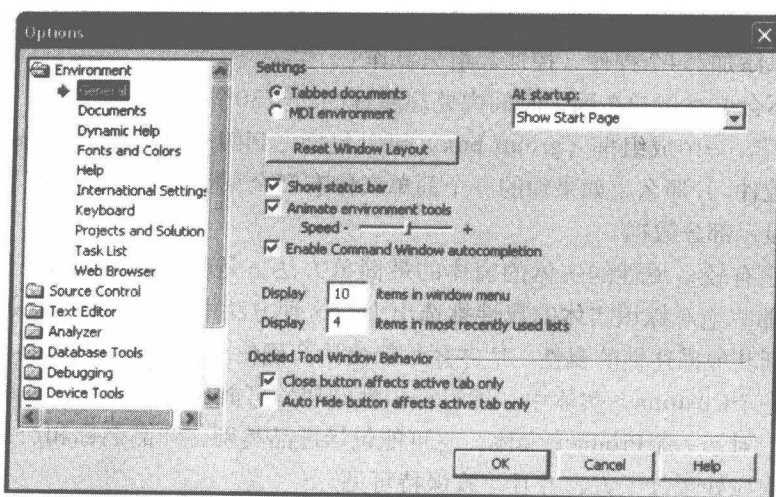


图17-2 Visual Studio中的面板架构是选项卡的一个很好的替代品

17.2 一对一联系

在大多数情况下，那些描述有一对一联系的实体信息的窗体可以和描述简单实体的窗体有着类似的处理方法。你可以创建一个查询，包含来自两个表的合适的字段，然后就可以将其视为一个简单实体来对待。如果因为属性较多而不能合适地在一个单一窗体上显示，那么可以使用前面阐述过的应用于简单实体上的相同技术。

如果主实体参与了多个一对一联系，那么这些联系的特性就决定了大多数自然的布局。如果这些联系可以共存，你可以将它们视为一个大的记录集而将信息显示在单个或者多个的窗体上，就如同我们在简单实体中讨论的一样。

但是，很多时候，联系都是相互排斥的，如同一个给定的Product要么是Beverage要么是Cheese，而不可能两者都是。这是一个使用多个子窗体或者面板的很好的机会，它提供给你足够的实用空间。你不必担心用户会认为在给定的环境中还有更多相关的信息；其实这里没有更多的信息了。事实上，在这种情况下使用选项卡控件将会带来误导。

如果你有实用空间问题，那么你需要使用一种技术来分组显示属性，该技术我们在简单实体中讨论过。如果子类中描述属性的控件被放置在一个选项卡中，那么最好是使用一个一般形式的选项卡控件。如果当用户浏览表格的时候，选项卡（或命令按钮标签）发生了变化，会使用户十分迷惑的。

同样的道理，如果用户能够浏览主要的记录集，我会设法避免在选项卡控件的第一页中添加显示子类实体属性的控件。由于用于每个子类的控件都各不相同，因此在用户浏览记录集时窗体的显示就会不断跳动变换。避免子类属性不出现在首个选项卡上，可以保持显示的稳定性，并且由于不用重复计算没必要的显示，也顺带地提供一些性能优势。

17.3 一对多联系

许多窗体需要显示关联在一对多联系中的实体。只要你记住窗体需要显示的是一对多联系而不是多对一联系，那么决定这些窗体的布局就很容易了。当你在为复杂的联系建模时，

如果根据记录来考虑而不是根据实体实例来考虑通常会简单很多。因此在这种情况下，你必须肯定在“一”端的记录控制“多”端记录的记录，而不是其他方式。如果试图用“多”端的记录来控制“一”端的记录，那么你将会把自己（还有系统和你的用户）交织在一起。

这是显而易见的，至少对我来说，但是它的的确确是一个普遍错误。很多年前，我曾花了一大笔钱来修正一位头衔为“高级界面设计师”的女士为一家澳大利亚银行设计的所有“多对一”的窗体。这是我第一次全然相信这么一个理论：“真正的程序员是不能为银行工作的”。

当已经使在“一”端的记录来控制窗体的显示之后，你需要决定如何显示“多”端的记录。可以有两个选择：一次全部显示或者一次显示一个。

你的选择在很大程度上取决于用户需要查看每条“多”端记录的细节数据量。如果仅需要显示一小部分字段，那么通常可以一次全部显示它们。图17-3显示的来自Northwind样板数据库的窗体，就是使用一个子窗体来显示“多”端的每个记录的四个字段。

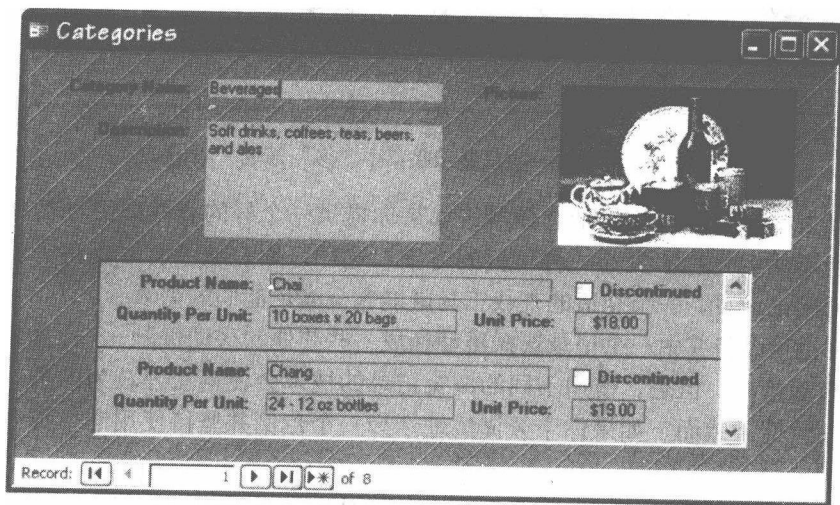


图17-3 该窗体使用一个连续查看的子窗体来显示联系中“多”端的记录

在该子窗体上的垂直滚动条说明不能一次全部显示出所有记录。出于设计的目的，这仍然算是“一次显示所有的记录”。在下一章，我们将关注其他一些显示多条记录的方式。

我喜欢这种“一次显示所有记录”的窗体布局，因为它能一次给用户提供大部分的内容。当然如果你确实有实用空间问题，你也可以提供一种使用一个次级窗体来显示附加的细节信息的方法。但是这种风格的布局并不总是合适的或可能的，而且你可能需要一次显示一条“多”端记录。这种情况下，你需要澄清两个问题：一是使你的用户清楚地知道这里有多条记录；二是提供一种能让用户在各记录间移动的机制。

图17-4显示的窗体与图17-3所示的窗体相同，只不过是在子窗体中显示了单条记录，而不是以一种连续的形式显示。

Microsoft Access在窗体的底端提供了一种默认的记录导航机制，那就是“播放式按钮”记录选择器。作为一种规范，任何可能的情况下，我都坚持这种默认的方式。但是看看图17-4显示的结果：其中两个窗体各自都有一个记录选择器在上面，仅仅只能通过它们各自所在的

位置来表明哪个记录选择器是属于子窗体的记录集，哪个记录是属于主记录集。不到穷途末路的时候，这就不再是最佳的解决方案。

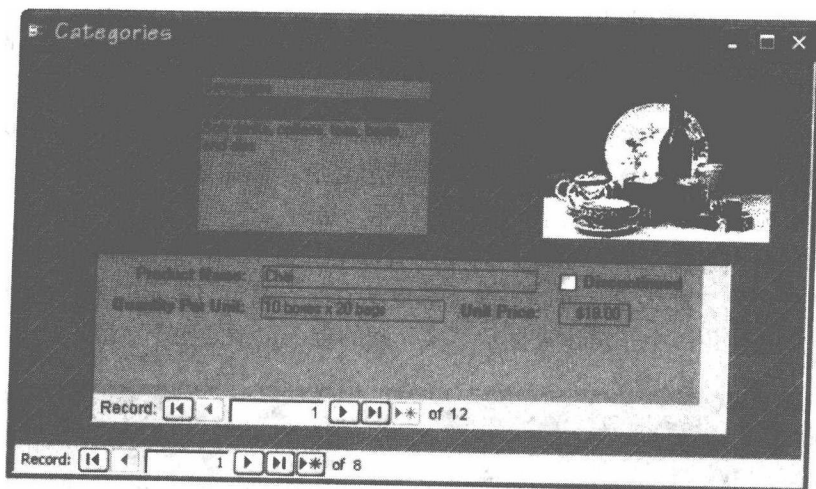


图17-4 该窗体一次只显示联系中“多”端的一条记录

有时可以去掉主窗体上的记录选择器。如果你提供一个好的查找产品类别的界面，那么对图17-4所示的窗体也不失为一种可行的方案。这样，用户就没有必要在一个如图所示的维护窗体中翻阅记录页，一个合适的查找工具就能完全满足他们的需求了。

当然，你也可以将其中一个记录选择器替换成一些其他的移动机制——比如命令按钮或者一个自定义滚动条等。但是，使用不同的浏览记录的技术会使得处理较为困难，因为你给界面引入了不一致性。然而，如果你十分谨慎地选择一个范例并坚持使用它，那么这种方式也未尝不可。

例如，你可以使用类似在Microsoft Internet Explorer中的Forward和Back工具按钮来控制“一”端记录间的移动，而使用播放式按钮记录选择器在“多”端记录间移动。只要你在应用程序中一贯使用这种形式，那么也是可行的。其实即使是简单实体，它们没有“多”端记录，也可以使用Forward和Back按钮。

偶然地，如果一个窗体将要显示一个有多个一对多联系的主表，那么你就需要显示多个处于联系中“多”端的表。

如果你采纳我的建议，那么在可能的情况下，你可以避免上述情形。多个“多”端表在一个窗体上会使得设计相当混乱，而且使得实现更加混乱，但有的时候你别无选择。如果你必须在一个单一的窗体上显示多个一对多联系，最简单的解决方案就是使用一个选项卡控件来分开“多”端记录的显示。这种解决方案给用户提供了一个清晰的环境。它也可以提高性能，因为选项卡控件上的记录只有在选项卡显示的时候才会被加载。

我们应当竭力去避免的是让全部的多记录控件同时可见，这样它们会变得很慢而且相当难看，并且它们在一个单一的窗体上出现，就隐含着“多”端实体的一个联系，但该联系很可能是根本不存在的。例如在图17-5中，所列出的电话号码到底是属于公司还是联系人，显然是不清楚的。

图17-5 窗体中的电话号码属于公司还是联系人是不清楚的

17.4 层次

任何一对多联系从技术角度来说都是一个层次，但是这个术语通常对含有三个或多个记录集的联系是较为严格了些，如果你愿意，你可以称之为一对多对多联系。在数据模型中，层次性的联系是经常发生的，但是它们通常没有必要在窗体中描述出来。

例如，在图17-6中，从Customers到Orders再到Order Details联系就是一个三级层次联系。

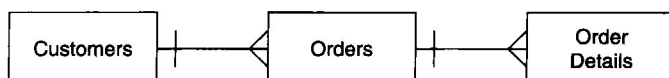


图17-6 该E/R图显示一个三级层次

在这种结构下，大多数应用程序都有一个Customers窗体，如果它一定要显示订购信息，那么会以概要的形式来显示。而使用一个单独的Orders窗体来显示Orders和Orders Details项。当然，这个Orders窗体会关联上Customers实体，但是，仅仅是为了显示Orders和Orders Details二者之间的一对多联系。极少数应用程序会要求单独使用一个窗体来显示来自三个表中的Orders Details细节信息。

但是，很难说这种避免显示层次性联系的倾向是由于确实缺乏必要，还是由于的确很难合理地显示它们。例如，可以猜想客户要求有使用Customers维护窗体来查阅订购信息的能力。一个简单地提供这项功能的方法就是一次显示Orders表（中间一级）中的一条记录，而同时显示所有的Order Details记录。本质上来说，就是在Customers维护窗体上添加一个如图17-3所示的窗体作为一个子窗体。

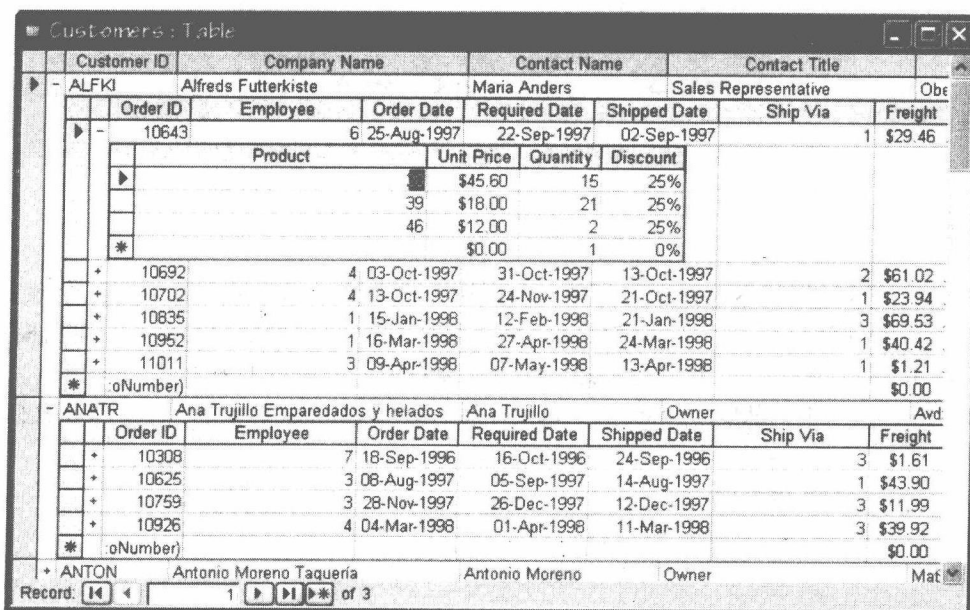
然而，在这种情况下，一次浏览一条Orders记录确实不太合适。用户很可能更倾向于询问：“这个客户到底订购了多少次”，或者“他的平均订购总和是多少”，而不是去问有关的具体产品。如果你一次仅描述一条Orders记录，那么回答以上这些普通问题将会变得相当无聊了。

为了避免这样的问题，你可能决定在Customers窗体上仅显示一些概要的订购信息，如果用户要求更多细节信息，则提供一种打开次级Orders窗体的机制。但很不幸的是，这种“细节概要”的方法也含有一些严重的缺点。

如果用户想要查看订购细节以便决定哪些产品是客户最常订购的，他们不得不耐着性子打开所有的次级窗体。因为次级Orders窗体通常在单一的窗体视图中仅显示一条订购信息，它很难用来比较在多种情况下订购的产品。

另一个选择是使用一个TreeView控件。TreeView控件是一种十分优秀的工具，但是它们在层次的每一级上所显示的数据量是有限制的，因为所有的数据必须出现在同一行上。这种局限性使得TreeView控件也不适合我们的例子。试图将客户的所有细节信息均显示在一行，的确是很难办到的。

Microsoft Access 2000使用了子数据表单来支持以略图的格式描述层次数据，如图17-7所示。



Customer ID	Company Name	Contact Name	Contact Title
ALFKI	Alfreds Futterkiste	Maria Anders	Sales Representative

Order ID	Employee	Order Date	Required Date	Shipped Date	Ship Via	Freight
10643		6/25/1997	22-Sep-1997	02-Sep-1997	1	\$29.46
	Product	Unit Price	Quantity	Discount		
		\$45.60	15	25%		
		\$18.00	21	25%		
		\$12.00	2	25%		
		\$0.00	1	0%		

Order ID	Employee	Order Date	Required Date	Shipped Date	Ship Via	Freight
10692		03-Oct-1997	31-Oct-1997	13-Oct-1997	2	\$61.02
10702		13-Oct-1997	24-Nov-1997	21-Oct-1997	1	\$23.94
10835		15-Jan-1998	12-Feb-1998	21-Jan-1998	3	\$69.53
10952		16-Mar-1998	27-Apr-1998	24-Mar-1998	1	\$40.42
11011		09-Apr-1998	07-May-1998	13-Apr-1998	1	\$1.21

Order ID	Employee	Order Date	Required Date	Shipped Date	Ship Via	Freight
10308		18-Sep-1996	16-Oct-1996	24-Sep-1996	3	\$1.61
10625		08-Aug-1997	05-Sep-1997	14-Aug-1997	1	\$43.90
10759		28-Nov-1997	26-Dec-1997	12-Dec-1997	3	\$11.99
10926		04-Mar-1998	01-Apr-1998	11-Mar-1998	3	\$39.92

Order ID	Employee	Order Date	Required Date	Shipped Date	Ship Via	Freight
10308		18-Sep-1996	16-Oct-1996	24-Sep-1996	3	\$1.61
10625		08-Aug-1997	05-Sep-1997	14-Aug-1997	1	\$43.90
10759		28-Nov-1997	26-Dec-1997	12-Dec-1997	3	\$11.99
10926		04-Mar-1998	01-Apr-1998	11-Mar-1998	3	\$39.92

图17-7 Microsoft Access 2000使用子数据表单来显示层次数据

子数据表单也不尽人意，但还能起到一定的作用。它最多可嵌套八层，但是在每一层只能显示一个单一的记录集。例如，你不可以创建一个子数据表单来在同一级中同时包含Addresses和Orders。

Visual Studio.NET支持一种DataGrid控件，它提供了类似的功能，但是使用起来也很不方便——虽然它能维护数据集之间的层次联系，但是它每次仅显示一条记录。在我们的示例中，一旦用户显示了Order表的数据，那么Customer数据就要消失，而用户需要使用一个Back按钮来重新显示它。

17.5 多对多联系

最后一种你需要在窗体中显示的逻辑联系是多对多联系，在数据库中它通常是由多个表来表达的。

在绝大多数情况下，你可以将多对多联系看成是一对多联系。例如，图17-8显示了Customers和Products之间的多对多联系，而Orders和Order Details表是作为连接表。

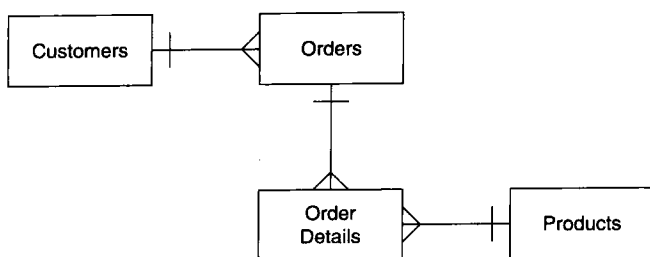


图17-8 Customers和Products有一个多对多联系

我们可能需要显示一个客户订购的所有产品（这种情况下，你可以将Customers表看作联系的“一”端），或者显示订购同一产品的所有客户（这种情况下，Products表处于“一”端）。你可以使用任何一对多窗体中的相同的表达技术，只需要处理在何处包含来自连接表中的数据，以及如何处理“多”端的重复值问题。

大多数连接表仅包含多对多联系两端的主码。但是，正如我们在第3章看到的以及图17-8所示的，联系本身有时也有一些属性，它们被作为连接表的一部分。如果你需要在窗体中包含这些属性，那么它们应当被显示在“多”端。

在我们的示例中，如果窗体将要用来显示每个客户的产品（Customers表作为“一”端）、订购日期——Orders表中的一个字段——显然是产品信息的一部分，而不是客户信息。“客户X在15号买了产品Y，在18号买了产品Z……”。如果是以另外一种方式来显示，而将Products作为“一”端，那么订购日期将会成为客户信息的一部分。“产品X在15号由客户Y购买，在18号由客户Z购买……”，这种情况下好比Customers表作为“多”的一端。

在“多”端的显示中显然很可能存在重复，或者至少是部分重复。你必须决定是否分别显示每个项，或是显示一个计算信息。例如，如果你要将购买每个产品的客户罗列出来，可以为每个产品一次显示购买它的所有客户；或者你可能仅仅一次列出一位客户，并显示出客户订购的总次数以及购买的总数量（也可能是平均数量）。

在这里，你需要十分仔细以避免罗列完全重复的项。如果你并不提供任何额外的信息，比如订购日期，那么简单地将一位客户的名字罗列27次是完全没有意义的。用户想看该信息的唯一理由是来决定一个客户订购的次数，但是应当由应用程序而不是用户来做这样的计算。

虽然将一个多对多联系看作一对多联系满足了大多数的要求，但是有时候还是需要你完整地显示该联系。例如，一位产品部经理查看那些购买特定产品的客户，同时还想了解这些客户还购买了哪些其他产品，以便实行“一揽子交易”。

幸运的是，这种类型的分析在一个多维数据库中是十分常见的，正如我们所看到的，多维数据库界面工具，如Microsoft的Data Analyzer，是特别为处理这类情况而设计的。但是这种在一个产品数据库中作为管理功能一部分的需求是相当正确的。

这种情况下，一个相对简单的解决方案就是将这种联系看作层次来对待，并使用子数据表单或者DataGrid控件来显示。这种技术的不利之处在于额外信息所描述的内容不够清晰。例如，如果你在一个产品列表中插入一个仅罗列客户名称的子数据表单，那么就无法立刻十

分清楚地分辨出这些客户的名字哪些是购买产品的，哪些是产品货源的供应者。

如果你在意一个层次显示的潜在的迷惑，或者如果客户一般并不要求层次信息，那么最好是在一个次级窗口中提供这些信息，这样就可以使它的意义更明确。使用次级窗口同时也允许你提供额外的总结信息或者替代直接的细节信息。

相对于一位客户购买的其他产品来说，产品部经理更希望看到所有客户均购买的产品列表，然后根据交迭的百分比将这些购买主要产品的客户组织起来——例如，那些购买widget的所有客户均购买了gizmo，但是其中仅仅只有10%的客户购买了doohickey。当然，你可以在主窗体上提供这些总结信息，但由于计算相当复杂（因此也很耗时间），除非是经常需要，最好不要在次级窗口中显示这些信息。

这两种显示多对多联系中所有数据的技术之间是相互排斥的。你可以在主窗体上用层次显示的形式提供每个产品的购买者的列表，也可以在一个次级窗口中提供一个“购买最多的十位客户”名单，用户可以根据需要查看。总之，你必须将你的决定建立在你期望窗体如何被使用的基础之上。

17.6 小结

在第16章，我们已经关注了如何基于用户实施的任务来组织你的窗体。在本章中，我们学习了如何在窗体上构建控件，这是基于它们所显示的实体的结构。

在构建窗体时所做的选择主要是由窗体上所表达的实体决定的，如果要表达多个实体，则还要考虑这些实体之间的关联。其次，窗体的结构将由要显示的属性的数量决定，最佳经验证明，在一个窗体中一次显示的控件或控件组最好不要超过25到30个。

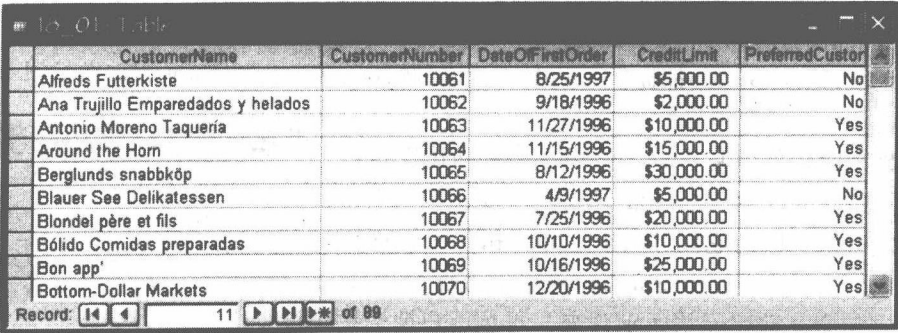
在下一章，我们将进入到用户界面设计的下一个层次：选择不同的控件来描述不同种类的数据。

第18章 选择Windows控件

在第17章，我们讨论了构建窗体的不同方式，这些方式都是基于窗体所描述的实体之间的联系。在本章，我们将关注怎样为特定类型的逻辑数据匹配特定的控件。

选择控件的基本原则通常有两个。最重要的一点是选择最能匹配用户考虑数据方式的控件——换句话说，就是匹配用户的意识模型。其次就是应当尽可能将用户的输入数据限制在最小的范围内。

使用数据库的用户倾向于将数据作为文本值来考虑。如果你在Microsoft Access的数据表中打开一个记录集，如图18-1所示，所有的字段均是以文本格式来显示的。但是事实上，只有CustomersName字段拥有文本数据类型。CustomerNumber 是一个随机数字段，DateOfFirstOrder是一个日期/时间类型，CreditLimit是货币型，而PreferredCustom是Yes/No（布尔型）。



CustomerName	CustomerNumber	DateOfFirstOrder	CreditLimit	PreferredCustom
Alfreds Futterkiste	10061	8/25/1997	\$5,000.00	No
Ana Trujillo Emparedados y helados	10062	9/18/1996	\$2,000.00	No
Antonio Moreno Taquería	10063	11/27/1996	\$10,000.00	Yes
Around the Horn	10064	11/15/1996	\$15,000.00	Yes
Berglunds snabbköp	10065	8/12/1996	\$30,000.00	Yes
Blauer See Delikatessen	10066	4/9/1997	\$5,000.00	No
Blondel père et fils	10067	7/25/1996	\$20,000.00	Yes
Bólido Comidas preparadas	10068	10/10/1996	\$10,000.00	Yes
Bon app'	10069	10/16/1996	\$25,000.00	Yes
Bottom-Dollar Markets	10070	12/20/1996	\$10,000.00	Yes

图18-1 该数据表将不同类型的数据均作为文本值来显示

当系统设计者操作这些数据时，我们当然清楚不同数据的值域，而且不会去尝试将一个日期和一个布尔值联系起来。尽管如此，依然有一种倾向是将不同类型的值用同样的方式显示，通常都是文本框。你当然可以在一个文本框中显示任何数据类型，但是这种工作方法对你的用户来说毫无益处。应该在确定其他任何更具体的控件类型都不合适的情况下，将文本框作为最后的选择。

比如一个声明为日期/时间类型的字段，你可能仅仅是看到一串特定形式的字符，但对于用户来说却是一个日期。改变一个日期和改变一个文本字符串，对于用户的意识过程来说是绝然不同的。如果一个用户敲错了一个名字，她可能会想：“Jary中的J应当是M”。然而，如果一个用户需要更改一个日期，那么她则更可能这样想：“那应当是自周一以来的一个星期”。你完全可以通过选择一个支持用户考虑数据的方式的控件来使用户的生活更容易一些。

同样地，你也可以通过限制他们输入的值来使用户的使用更简易。数据限制和数据有效性是两个不同的问题，我们将在下一章讨论数据的有效性问题。数据有效性是系统实施的事后检查，以保证已经输入的值是合理的。而数据限制是在输入之前来防止用户输入不合理的

数值。

当然，首要的是限制用户输入的数据类型。为了选择合适的控件，可以将数据分为四组：逻辑数据、数值集合、数字和日期以及文本。我们在这一章中逐一讨论它们。

注意：我不可能无止境地讨论来自不同的第三方供应商的不同产品，我将我的探讨限制在由Microsoft Visual Studio.NET和Microsoft Access所提供的产品上。但是，我十分鼓励你去熟悉一个自己可用的工具。（微软的网站是一个很好的去处。）你可能在上面找到很多垃圾产品，但是的确也有很多相当不错的工具。这样你可能会发现花几百美元购买一个第三方工具可以为你节省数周开发和测试的时间。

18.1 表达逻辑数据

虽然你可以将逻辑数据显示在一个文本框中，但这的确不是一个好的方法。例如，一个Customer表可能包含一个声明为布尔值的CreditApproved字段。你可以使用一个文本框来显示该字段，然后使用数据有效性检查来确定用户输入的是“Yes”、“No”，“True”或者是“False”。但是如此允许用户无限制地输入数据，就好比请求用户输入“临时的”或“欠账的”一样。除非你准备好了接受和解释这些实体，那么请你为你和你的用户考虑一下——选择一个限制为两个值的控件。

Access和Visual Studio都提供了两个适合这种情况的控件：复选框和触发按钮，如图18-2所示。

大多数人比较熟悉复选框，使用它们来描述逻辑值是一个不错的选择。触发按钮相对使用的较少。它们对布尔值更加有效，因为它们等同于“on”和“off”而不是“true”或“false”。触发按钮的问题是在它处于“off”状态时，它与命令按钮无法区别开来。既然用户期望按下一个按钮就会触发某些活动执行，很多人可能在按下一个标记为“CreditApproved”的按钮时就会十分犹豫

了，他们会想这么做会初始验证的过程而不是简单地标明验证过程已完成。鉴于这个原因，我倾向于将触发按钮成组使用，如同单选按钮一样。事实上，这正是Visual Studio中所实现的。他们是RadioButton控件，其外观属性被置为“Button”。

关于单选按钮（也被称为可选按钮），请不要用这种控件来描述单一的逻辑数值。虽然任何Microsoft Windows环境下都没有禁止这样的方式，但是这是没有必要也不合适的（复选框同样也是）。单选按钮适合用来描述一组相互排斥的选择。一个单一的可选按钮看上去相当的孤单。

更糟的是，如果你在窗体的同一区域使用多个可选按钮，那么用户很可能错误地认为这些按钮之间有某种关联，并且他们可能同一时刻就只选择一个可选按钮了。此外，他们会期望可以自动地选择一个选项而使其他选项无效。当一个计算机系统没能按照用户期望的形式运行时用户总是会因此而不安。

18.2 表达多个值的集合

很多控件允许你在一个窗体上显示多个值。这样的选择主要取决于你需要获取单个值

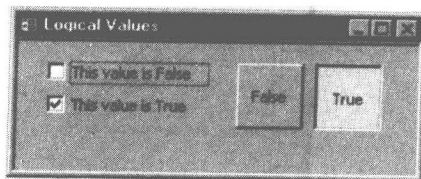


图18-2 每个复选框和触发按钮都可以用来显示一个逻辑值

(尽管也是从一个范围的值中选择)还是多个值。

18.2.1 从一组值中获取单个值

一个给定的值是否被接受通常由它是否存在于某个列表中决定。例如,当且仅当一个客户编号存在于Customers表中时,Orders记录中的该CustomerNumber值才有效。假如你想要在任何可能的地方限制用户在可接受的范围内输入数值,那么你需要将Customers列表展现给用户并允许他们从中选择。

在这种情况下通常选择的控件是组合框和列表框,如图18-3所示。这二者在功能上有所不同,最重要的一点是组合框允许用户输入不在列表中的数值。正如其他地方所讨论的,该功能有时可以被用来为其所链接的实体创建记录。即便在特定环境下不适合输入新的记录,这种通过直接键入来查询一个数值的方式的确受到大多数打字员的欢迎,他们认为伸手拿鼠标,甚或看着屏幕都是十分不方便的。

Visual Studio允许组合框被设置成总是显示成列表形式或者仅仅在需要的时候显示。Access仅允许使用下拉组合框。(此外,Visual Studio中的列表框可以在每个值旁边显示复选框;Access则不支持这样的功能。)

《软件设计的Windows界面指南》一书阐明了你应该将简单组合框和列表框的大小设置为显示3到8个项。如果你在窗体上找不到足够的空间来合理的放置它们,那么你应当使用一个带下拉列表框的组合框。

即使你有空间来永久显示该列表,你也应当使用一个下拉组合框以防一旦用户做了某个选择而使得列表不相关了。这时,允许用户浏览一下哪些值可用是非常有效的方法,特别是当列表本身或者记录中的值频繁变动的时候。

例如,在一个图书馆系统中,标题被设为一个分类主题而形成一个列表,该列表需要时常更新和整理,用户可能想要查看最新的列表,以决定是否为一本给定的书定义一个新类别。

但是比如你在一个Orders窗体上提供一个客户的列表,一旦用户选定了他们想要的客户,他们不会在意系统是否识别其他的客户。这种情况下,当用户选择完毕后,需要使用一个下拉组合框来隐藏该列表。

在Visual Studio和Access中有一个十分有用的功能,它能显示一个不是表中存储的数值。例如,如果你在Customers表中使用一个计数值或者标识值作为主码,你需要将其作为Orders表的外码存储起来,但是没有理由让用户看到这个值。在Orders窗体中,你可以很容易地显示客户名称(来自Customers表),以使用户来选择。

在Visual Studio中,可通过为DataField和ListField属性指定一个不同的值来实现这个功能。在Access中,可使用多个列——一个是客户名称,一个是客户编号——将该控件绑定到Orders表的客户编号列,并且通过将其宽度设置为0来隐藏客户编号列。

Access组合框显示多列值的能力十分的有用,我觉得十分遗憾的是在Visual Studio中却没有。例如,在显示一系列客户时,如果能包括一些额外信息,比如他们居住的城市等,会显得

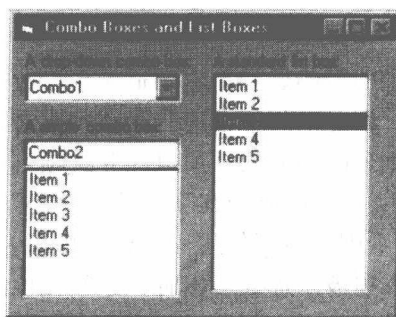


图18-3 Visual Studio提供了三种形式的组合框

更完善，能进一步标识客户。在Visual Studio中，你只能通过使用一个计算字段链接到这些额外信息，并作为ListField属性来设定。从某方面来说，这绝对没有Access组合框好用，特别是当有些值为空时。

虽然列表框和组合框十分有用，但是如果列表中包含了过多的数值，那么它们也无能为力了。如果有成百项数据，那么就需要寻找其他的途径来限制列表。可能的话，可以让用户来选择一个字母表中的字母、一个销售地区或者居住的州，然后在以上选择的基础上来过滤列表项。

如果列表非常小——不超过5至6项——并且它的数据值是固定的，那么你可能想使用一个选项组而不是组合框或列表框。可以使用可选按钮或者触发按钮来实现选项组，如图18-4所示。可选按钮是更通常的选择。

但是，尽管这是可行的，但是至少在Visual Studio中，在运行时生成一个选项组并不是个明智的选择。它不适用于用户改变窗体的布局，因为如果你在运行时添加或者删除选项将会十分的不方便。所以除非非选项列表是固定不变的（或者至少在下一个版本之前是不变的），否则你最好使用一个列表框或者组合框。当列表项被添加或移出时，这些控件的大小依然会保持不变，因此对于用户来说会十分方便地修改列表项。

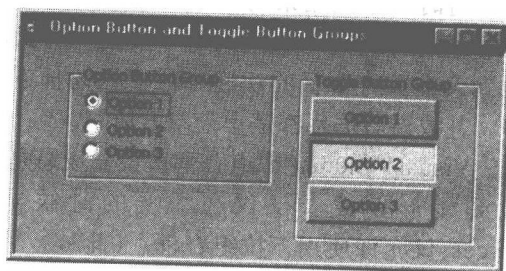


图18-4 在一个成组框中的可选按钮或触发按钮可显示短小确定的列表

18.2.2 获取一组值

如果你需要获取一组值，那么你操作的是一个一对多联系的“多”端。正如我们在第17章所看到的，在这种情况下，你需要作的第一个决定是你希望一次显示和获取所有的记录还是一条记录。

如果在Access中，你想一次显示和获取一条“多”端记录，那么可以在单条记录视图使用子窗体来实现。设置LinkChildFields和LinkMasterField属性，Access会为你做大多数工作。在Visual Studio中，创建一个子窗体需要多一点工作（好了，是很多工作），但是会让你有更多一点的控制。不管哪种方式，对你要获取多个字段值来说都是一个很好的技术，并且特别是当你想要使用不同的控制方式的时候。

如果你需要选取每条记录的多个值并且想同时显示多条记录的话，可以使用Access中的数据表单或者Visual Studio中的DataGrid控件。Access数据表单是十分有效的（有时过于有效），它们允许多种控制方式来添加文本框。事实上，Visual Studio的DataGrid控件实现起来有些费劲，但是依旧可用（相对早期的Visual Basic版本中的Grid控件要稳定很多）。

如果在窗体显示时，你希望能更好控制窗体，Microsoft Access提供了另一种方法。Access的子窗体控件支持在连续的窗体视图中显示子窗体，这样就可以同时查看多条记录。如果你需要显示多条记录并且仅使用一种显示方式——比如一个选项组——不使用数据表单和表格控件，那么这无疑是个好的选择。为了在Visual Studio中实现同样的功能，你需要创建一个用户控件。

在某些环境下，另一个比较适合显示多条记录的控件是TreeView控件。TreeView控件常被用来在窗体轮廓中显示层次数据，但是它也能用来显示每条记录被选中的细节数据。TreeView对于编辑这些细节数据不是一个有效的机制，但是对导航记录来说是十分有用的。使用一个类似Microsoft Windows Explorer的范例，你可以根据在窗体的另一部分所选择的记录来显示细节数据。

最后，当用户需要从一个列表中选择一组项的时候，一对相互联系的列表框会是一个很有用的结构。如图18-5所示的Sample Fields和Fields In My New Table两个列表框，这种结构经常用在向导中。（屏幕上所显示的就是Access 2000的表向导。）用户很容易理解这种相互联系的列表框方式，但是需要注意的是，这种方式实现起来比较困难。

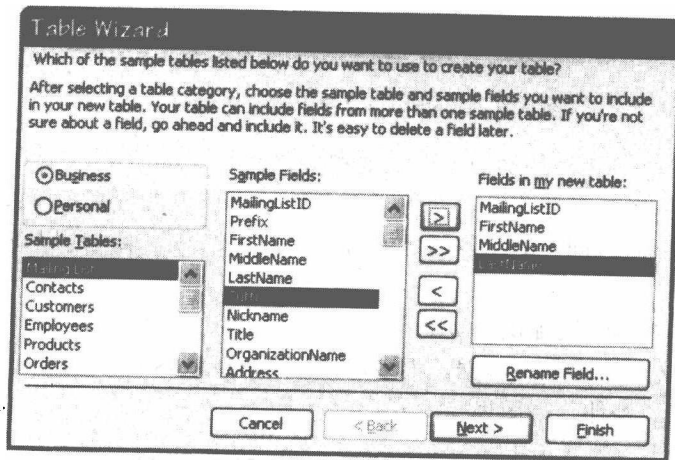


图18-5 Microsoft Access 2000的表向导使用相互关联的列表框来允许用户为新表选择字段

像这样的一对列表框在初始化数据输入的时候相当方便，但是，如果没有其他理由让它占用太多空间的话，在接下来的编辑和显示中它就不是一个合适的结构了。这就是为什么这种技术大多只用在向导中，这样每次数据输入都可以在一个单独的屏幕中处理。一旦初始化数据完成，用户很可能就不再需要参考这些已选择的和未被选择的完整列表了，并且你能更有效地使用前边列出的某个控件了，或者甚至只使用一个多选列表框。

但是，由于多选列表框的选择工作方式，这种做法会比较危险。对于用户来说，只需要单击一个新项（而不是点击控件），就可以十分容易地突然取消所有当前已经选择的项。当这些选项是与数据绑定的时候，挑选出记录的添加项、删除项和复选项是一个相当可怕的解决方案。一个较好的解决方法是使用一个通常的单选列表框仅显示那些已被选择的项，可能还需要一个文本框或者一个组合框来添加这些项。

18.3 表达数字和日期

数字和日期通常都在文本框中描述。和通常一样，在可能的情况下最好约束用户只能输入数字型的值。但也不能允许输入的值的范围太大。

Access中的输入掩码属性和Visual Studio中的MaskedData控件针对文本框输入方式提供了一定的控制功能——你至少可以禁止字符数据的输入。Access也能较好地控制数据的输入，但

是这毕竟是一种事后的数据检验而不是最好的数据限制方式。

Visual Studio提供了额外的输入日历数据的控件，MonthView控件和DateTimePicker控件，如图18-6所示。Access提供了一个类似Visual Studio中的MonthView日历控件。

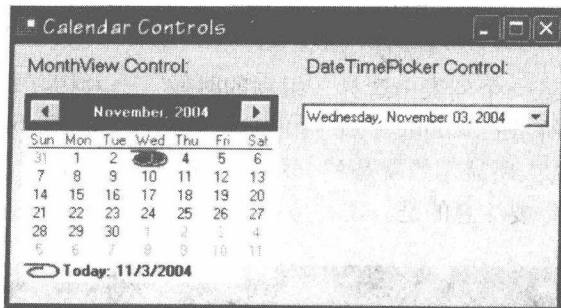


图18-6 MonthView和DateTimePicker控件

你可能会认为这些日历控件类似于列表框和组合框——DateTimePicker仅在需要的时候显示日历，而MonthView一直都会显示。但是两个控件都仅仅处理日期，而不是日期和时间。这就导致当它们与日期/时间型的字段绑定的时候会有一些困难，所以在实现的过程中要十分注意这一点。

Visual Studio和Access还提供了Microsoft UpDown控件（有时被称作微调控件），它可以被用来输入数字型或日期/时间型数据。很多用户都对Windows中的日期和时间设置较为熟悉，它就是使用了UpDown控件，如图18-7所示。

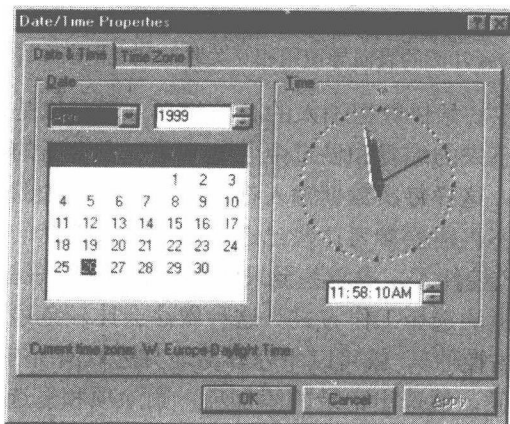


图18-7 Windows使用UpDown控件来控制时间值

如果值不是均匀增加的，UpDown控件会特别有用——例如某个周日的日期，或者一个最接近百的数字。

最后一组可用来输入数字型数据的Visual Studio控件是Slider控件和Scroll bar控制，如图18-8所示。

这些控件在数据库应用中有一个清楚的受限的值。它们主要就是对于设置的数字值以图形化的方式显示，因此它们从理论上来说可以被用来输入任何数字数据。（你很可能不认为滚动条

是一个数字型工具——那不是它们的表面模型——但是它们确实给应用程序返回的是数字值。)

我认为，当用户需要比较一些相对位置比其精确值更重要的数据的时候，这些工具就相当有用了。可以这么说，“这个字段的值比那个字段的值更重要”。例如，我偶尔使用滑动条来匹配检验，允许用户设置相对重要的或者要求公差的字段来比较，以便为某个特殊的记录找到一个匹配值。

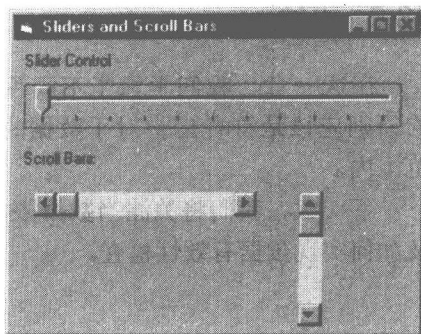


图18-8 滑动条和滚动条在数据库中应用中提供受限的值

18.4 表达文本数据

最后，我们来看看那些“不得以”的控件——文本框。文本框是无辜的，只不过它们被过度使用了。一个典型数据库中的大多数数据都是文本的，并且文本框是描述这些值的比较合适的控件。例外情况是当字段是一个外码时，这种情况下如前文所说，组合框或者列表框通常是一个更好的选择。

同样地，对于文本框的使用，最根本的控件选择原则是：在符合用户意识模型的情况下使用它们，并且尽可能的采用数据限制技术。

在Access中数据限制技术是由文本框控件的Input Mask属性来提供的，在当前版本的Visual Studio中并不直接支持这个属性。输入掩码允许你指定一个特定的模式来控制数据的输入格式。例如，美国社会保障号应当遵循这种模式：###-##-####（三位数字，一个破折号，接着两位数字，一个破折号，最后是四位数字）。

在某些特定的情况下，输入掩码是十分有用的。但是这些情况比人们想象的要少很多，用户更希望简单地限制数据输入的范围。电话号码就是一个很好的例子：你很容易想到为这些值使用一个输入掩码，但是在这之前，你需要确保系统只能接受来自同一个国家的电话号码。因为不同国家的电话号码常常有不同的格式。即使是在同一个国家，你也需要考虑可能的不同情况——例如，分机号和直拨号，或者简单的就是不同的约定格式。（过去时常让我很头疼的是在澳大利亚的一些人将七位号码格式化为999-9999，而另一些人则使用99-99-9999的格式。）

请记住，你提供一个输入掩码的目的是使用户的操作更方便。如果阻止他们输入完全有效的数据——不是例外，那么掩码就成为了一种阻碍了。

除了标准的文本框，Access和Visual Studio都支持使用一种Microsoft Rich Textbox控件来输入组合了不同字体和样式的文本。相对于使用其他控件来说，实现Rich Textbox控件要复杂很多，因为你必须同时提供一个设置文本属性的界面。

除了在界面上增加了复杂性之外，Rich Textbox控件也给数据的操作带来了一定的复杂性，这是由于将文本的格式化添加到了控件中。鉴于这些原因，我建议仅仅当这些附加的功能对用户来说是十分需要的时候才使用Rich Textbox。此外，应当只有在文本值被作为一个文本“块”时才使用它们，也就是这些文本虽然在不同的地方显示，但是不能用其他的值来查找或链接。

比如，“rich text”文本在管理样板图像文本时会十分有用，它们被组合用来制作标准信件。这些被包含的实际图像被存储为Rich Text的格式，但是那些用户选择的描述性的详细信息都以标准的、非格式化的文本存储。

18.5 小结

在这一章，我们学习了不同种类的控件，你可以用它们来显示和获取信息。选择控件的原则应该是符合用户对于数据的意识模型，并且尽可能将数据的输入限制在一个有效的范围内。

下一章，我们将关注当数据限制不可能实施时，该如何应对。同时我们还要学习何时以及如何实现数据有效性检查。

第19章 维护数据库的完整性

想象你是一个小的制造公司的老板，你已经十分努力地从一个大规模的竞争者那里吸引客户，并最终使得客户给你一个尝试的机会，但是你必须要在48小时内送出货物。于是你与制造商协商而且他们能够做到（通常极少可能做到），因此你开启了香槟。

可10分钟后，麻烦来了。在客户的信用卡被确认之前，会计师不能签署该订单，而这需要花一周的时间，如果没有一份签署的订单，制造商是无法进行生产的。你会怎么做呢？你很可能开始和会计师解释这种情况需要通融一下，改变一下规则。如果这不起作用，你会在完成订购前要求检查信用卡是你订的规则，并且因为这是你的公司，所以你可以在你认为合适的时候打破这种规则。如果都不起作用，你可能会解雇这个笨蛋会计，而亲自处理有关文件的工作。

我有理由相信那个会计和我们大家一样都是有家庭和债务的普通人，所以上述我所描述的情景是不太可能发生的。人们不会直白地拒绝老板的命令（或者至少不会经常拒绝）。但是计算机系统却通常这么做。它们会一边跺脚，一边伸出它们的下嘴唇，拒绝服从绝对合理的请求。所有这些可称之为“维护数据完整性”。

在本书的前面，我已经告知你什么是数据的完整性，怎样构建它以及在何处实现它，现在我要告诉你一个十分可怕的事实：维护数据完整性并不重要。

我现在暂停来等待抗议的呼喊声慢慢消失。

我并没有说你应该消除数据有效性检验，我只是说维护数据库的完整性远没有帮助用户完成他们的任务重要，并且你应该根据这一点来设计你的系统。数据确实需要最终有效，但是没有必要在它输入的时候就绝对有效。

让我们暂停一下，想一想为什么开发你的数据库系统：为了帮助人们完成一些任务。那些帮助人们完成任务的数据的有效性检验是很好的方式，它支持系统的目标。而有些数据有效性却阻止人们完成那些对他们十分重要的任务，或者阻止一些完全合理只是未曾预料的事情，这样的有效性检验都是很糟糕的。事实上就是这么简单。

如果仅仅是因为数据不完整或者你没有预料到，那么系统是不应当阻止用户输入数据的。当然，像大多数事情一样，这件事也是说比做起来容易。本章我们将关注不同类型的完整性约束，并且学习如何在意外错误的时候权衡地保护数据，从而维护系统的可靠性和可用性。

19.1 完整性约束的类别

在第4章，我们将完整性划分为六种，划分的依据是它们在关系模型中的逻辑层次。在这一章，我们将使用一种不同的组织方式。我们将完整性约束分为两类：内在约束和业务约束。

内在约束管理关系模型中数据的物理结构。如果在Orders表中存在Customers表的相关记录的话，系统会禁止用户删除Customers表中的这些记录，因为参照完整性是关系模型的一项功能，这条规则就是一种内在约束。如果允许用户在没有删除Orders表中相关联的订购记录的情况下删除一条客户记录，那么该数据库是不稳定的。如果之后添加了一个新的客户记录，

而它与刚被删除的客户记录有着相同的主码，那么那些孤立的订购记录将和这条新的客户记录关联起来，这显然是不正确的。这种情况是很容易发生的，比如你是从客户名字中派生出主码值的。

即使该主码值不被重用，那些孤立的记录也会造成数据库返回一些可疑的结果。例如，如果查询统计一个给定时期内的产品销售总数，那么根据Orders表是否与Customers表相关联，系统返回的结果是不相同的。为了详细列出销售给每位客户的各种产品数量，Customers和Orders表通常是使用一个自然连接而关联起来的。只有当Orders记录在Customers表中有相匹配的记录时，它才会被包含到销售总额的计算中。孤立的Orders记录在Customers表中没有相关联的记录，因此它们会被排除在计算之外。而另一方面，为了统计销售额，也常把Orders表和Products表连接起来，而那些孤立的记录则又会被包含到总销售的计算中。因此系统将提供两个不同的答案给这样一个问题：“我们在六月份销售了多少”。依据不同的措词将会有不同的答案，这显然是不能接受的。

另一方面，**业务约束**（也被称为**业务规则**）来自于问题域。只有当所有相关的订购记录被填写或者被取消的时候，系统才会允许删除关联到Orders表的一条客户记录。这样的规则就是一条业务约束。在这种情况下，业务约束会说“我们不能这么做”，而内在约束会说“这是不允许的”。

实际上，内在约束和业务约束的区别通常是不明显的，也是不重要的。重要的是其中一类数据完整性约束是来自问题域。你可以将这些约束当作给用户的一种便利来实现，并且如果这么做的确可以方便用户的话，你可以完全忽略这些约束。删除一个仍旧还有订购活动的客户显然是不切实际的做法——它很可能成为一个灾难。虽然一条业务规则说每位经理只能有五位雇员，但添加第六名雇员可能是一件便利的事情。事实上，不能输入那位雇员将会变得十分的不便。

关键是过分遵循业务约束将会损害数据库的稳定性或可靠性。如果一个用户删除了一条客户记录同时也删除了与之关联的重要的订购记录，那么系统会丢失重要的数据，但任何剩余的数据不会受到威胁。相关的客户记录和订购记录可以被重新存储和输入，那么就万事大吉了。

系统必须区别对待两类不同的数据完整性约束——内在约束和业务约束。内在约束不能被忽略，以便不危害数据的可靠性。而业务约束某些时候应当允许用户的明智判断来打破它。我们将在接下来的小节中进一步探讨每一类完整性约束的细节。

19.2 内在约束

这一类有效性规则我们称之为内在约束，它控制数据库的物理结构。这类约束包括管理数据类型、格式和长度的规则，是否接受空值的规定，范围约束，以及实体完整性和参照完整性。

19.2.1 数据类型

假定你已经选择好合适的控制方式，用户一般是不会与数据类型的约束相冲突的。我从未听说过有人会故意在Amount Due字段内输入一个日期，或者在一个复选框中输入文本。然而，有些人可能试图在一个接受数字值的文本框中输入“eleven”，这就是为什么数据输入控

件的选择是如此重要了，这一点我们将在最后一章中看到。

19.2.2 格式

格式化通常不会造成什么问题，特别是如果你能够在用户离开该字段后重新格式化输入（Microsoft Access和Visual Studio能够比较容易地做到这一点），或者提供一个输入掩码来指导用户输入。但是，要注意不要将数据格式设置得过于严格。如果输入的数据不能精确地符合你定义的格式，你最好尽可能去掉该数据格式的匹配方式，并允许用户按他们认为合适的格式来修订它。如果有效的格式意味着有效的数据的话，你就不必这么做了，但是这种情形是很少的。用户输入了一个形式为9-9999-99999-99的电话号码，可能会被一个糟糕的新电话系统简单地处理了。

19.2.3 长度

长度约束常常会造成问题——特别是那些字符字段的长度约束。不论你曾经有多慷慨大度，一个由于过长而无法匹配的有效数据总会出现你面前。有时候你可以通过使用字符字段并给它分配最大的字段长度（255个字符）并设置该字段为可变长类型，来避免长度约束的问题。在Microsoft Access中，所有的文本字段都是可变长类型。在SQL Server中，你必须明确地设置该字段类型为VARCHAR。这两个引擎都是仅为存储的字符分配空间，因此不会浪费空间。

当然，可变长字符字段并不是适合所有的情形。首先，有时可能会要求某个特定的长度。例如，社会保障号总是9位长度的。如果一位用户有一个10位的社会保障号，那么长度不是问题，而是数据无效。允许这种数据被输入显然是不明智的。其次，由于SQL Server处理包含可变长度字符字段的记录的更新，因此会导致性能下降。在大多数情况下，这种性能的问题可以被忽略，但是在性能要求很严格的应用程序中，如果频繁地进行数据更新操作，最好是使用固定长度的字段。（在添加数据时没有性能差别，仅仅是更新操作时有差别。）

最后，虽然允许十分长的字段长度可以提高可用性，但情况并不总是这样。事实上，有时候它会严重地破坏可用性。这种情况下，屏幕显示和报表格式都会十分难看（特别是报表，因为不可能在它上面滚动数据），此外查找包含特定信息的记录将会十分困难。

当不适合允许长数据值时，试着提供一些规范和规则来处理那些不符合的数据。如果一个客户名称确实超出分配的空间，你可以和用户协商来建立合理的规范来缩短它们，比如去掉诸如“The”等词汇，将“Company”缩写为“Co”，并且删掉“Company”之后的所有字符。这将会帮助保证——但不能确保——如“The Really, Really Long Name Company, Incorporated”之类的名称将总会被输入为“Really, Really Long Name Co”，而不是某次输入“Really, Really Long, Inc”，下一次输入“Really Long Name Company”。

19.2.4 空值

缺失值是用户在内在约束中时常遇到的另一个麻烦问题。我们已经在其他地方尽可能详细地讨论了空值问题，并且我已经说得相当清楚，我认为如果在现实世界中有任何数值未知的情况下，那么空值都应当被允许。至少这么做不会使得数据与系统完全无关。然而，如果你选择忽略这个明智的建议，就必须考虑怎样使系统支持用户试图输入尚不完整的一条新记录。

这样的问题发生在数据急需的时候。从分析系统的工作过程可以得知，一定数量的信息

需要在任务完成前获得。但是这并不意味着当记录第一次被创建的时候，提供所有任务需要的全部数据。你需要新雇员的银行账户的细节，然后你才能付给他薪水。因此账号细节字段在薪水发放日是不可以为空的。但是当用户首次创建这个记录时，数据库系统不应该仅仅是因为雇员手头没有足够的信息，就阻止用户输入该雇员其余的细节信息，也不应当因此妨碍其他任务的执行。用户可能需要制造一个门禁卡以便人们可以进入大楼。在该任务中，账户细节显然是不需要的。账户信息最终是需要的，但不是在上班的第一天就需要它。不要让你的系统发脾气，所有的事情都会按时完成的——雇员会留心看着它的。

如果你不愿意接受空值，哪怕是临时的，最简单的处理内在约束的办法是提供某个合理的默认值。系统中的一种可能性是在数据库模式中声明一个单一值作为默认值。事实上，系统能够给用户描述多个这种值的可选项——可能是“UNKNOWN”、“NOT APPLICBLE”或“YET TO COME”。有时候，系统能够在运行过程中计算一个合理的默认值。

19.2.5 范围

除了空值之外，特定的范围是另一个用户时常遇到麻烦的属性级别的约束。某些范围约束在数据类型中是十分明确的——例如，短整型数据类型存储的最大值只能为255。当数据类型来决定一个范围约束时，你就不需要给用户过多解释该约束。如果需要更大的值，你必须在数据库模式中定义一个更大范围的数据类型。我不推荐在应用程序的运行过程中执行这样的操作。

但是，如果已经在模式中将范围约束作为有效性规则或CHECK约束定义了，又或者是在应用程序而不是在模式中实现了该约束，那么这种类型的约束更像是一个业务规则而不是内在约束，如此一来你会有更多的余地来操作它（我们将在下一节中探讨业务规则）。

19.2.6 实体和参照完整性约束

你一定还记得实体完整性约束是保证表中的每条记录都是唯一标识的，而参照完整性约束是禁止记录引用那些不存在于本表或其他表的记录。你必须考虑如何处理实体完整性和参照完整性，而不是没必要地强加给用户。它们大多数是通过允许用户在一个存放有效项的列表选取值来实现的。但是，如同我们在上一章所看到的，这通常是不可能的，主要是因为列表会由于过长而没有操作性。

当不可能或者不适合将用户的输入仅约束在一个列表中时，就需要在数据刚刚输入的时候进行检查并给用户提示出现的问题。在数据库中，如果出现一条输入的记录与数据库中现存的记录重复，那么这便是一个实体完整性约束问题。系统的最佳反应是给用户显示那条已存在的记录——或者仅仅是相关的字段——让用户决定这条新记录是否是一个重复项，如图19-1所示。

注意当你显示已存在的记录时，不要覆盖了用户已输入的新记录的数据。将已存在的记录显示在一个单独的窗口中，并且允许用户决定是否重写这个新记录。也要注意，图19-1所示的对话框允许用户可以继续输入数据，而不用查看可疑的重复项。用户可能已经知道存在其他的记录，因此除非十分必要，否则就不需要打断他了。（跟我重复：“计算机不是万能的，计算机不是万能的，计算机……”）

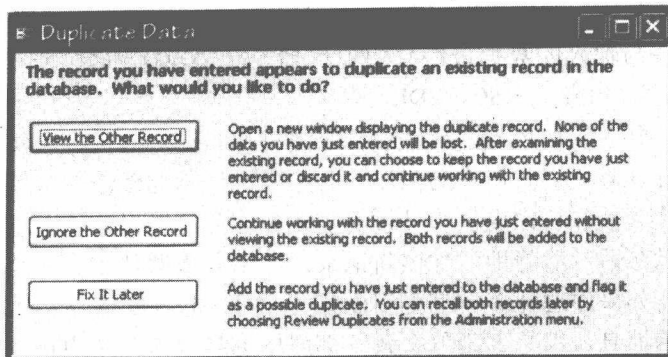


图19-1 该对话框允许用户决定新记录是否是一个重复项

那些试图让主码字段为空的用户提出了一个稍微不同的问题。主码字段的值不能为空是我所知的支持使用系统分配的主码值的最好论断。使用一个随即数字段（SQL Server中的标识字段）来保证用户可以对这个自然主码字段做任何他们需要的操作，而不必向实体或参照完整性约束妥协。如果在你的系统中不方便使用随即数（或者标识）字段，那么最好确保用户在每个主码字段中都输入某个值。显然简单的默认值是不起作用的，因为一张表在一次只能接受一个默认值。你可以让系统在运行中计算出一个值，但是如果你打算使用一个任意值，那么可能也会使用一个随机数字段。唯一的其他选择就是向用户请求一个可以接受的值。

当一个参照完整性问题发生时，通常是因为用户试图去关联一个不存在的记录。这可能是偶然的——例如，该用户可能拼写错了一个名字——或者也可能或多或少是有意的。该用户也许没有意识到那条关联的记录不存在或者可能还没有想到输入关联的数据。图19-2所示的对话框显示了处理这种情况的一种方式。这个对话框给用户四个选择：第一，他们可以现在就在系统中添加一条新记录以及相关细节信息；第二，他们可以在系统中输入一条新记录然后再来更新这个记录；第三，他们可以改变这个关联；第四，他们可以以后修订关联关系。

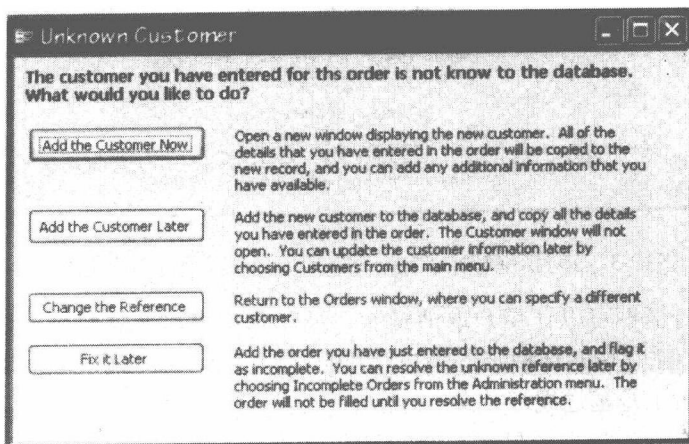


图19-2 该对话框允许用户选择怎样处理一个不存在的关联记录

注意，越过约束不是用户应当选择的。这些约束是不能越过的，因为正如我们所知的，这么做会造成数据库的不稳定。

如果是因为列表太长而无法为用户在一个对话框中显示出全部的有效项列表，那么你的对话框应当显示那些与输入的数据最匹配的项目列表。实现这种类似搜索的算法已有不少，比如简单的SQL LIKE语句或一个SOUNDEX搜索。

在某些情形中，根本没有必要显示一个参照完整性检验对话框。事实上，通常最好避免这么做，因此也就可以避免打断用户的工作。如果你有足够的理由确信用户想在添加了新记录之后再更新细节数据，并且你提供了足够的功能在你猜测错误的情况下取消系统的相应操作，那么系统就可以在后台静静地添加该新记录，而不必要做什么额外的检验等操作。在用户完成该记录之后，如果你想让该信息更明显些，你可能想要在状态栏中显示一条消息或是在一个对话框中显示消息。但是不论怎么做，都应尽可能避免在用户正在输入数据时打断他们。这是没必要的粗鲁并且会吓到用户的行为。

一些轻率的用户会尝试去关联一条不存在的记录，或尝试删除或修改一条被关联的记录。比如，用户可能会删除一个有着显著订购记录的客户信息，或者修改一种被关联在OrderItems表中的产品的ProductID值。

Microsoft Jet数据库引擎支持级联更新和级联删除，它允许更新和删除关联的记录，而不需要用户干预处理。你可以在SQL Server中使用触发器来实现同样的功能。如果级联更新和删除在你的应用程序中奏效，那么使用它们将是迄今为止最好的解决方法。但是，如果只有用户能决定最好的操作方式，那么就必须为他们的每一个选项解释其中的含义，如图19-3所示。

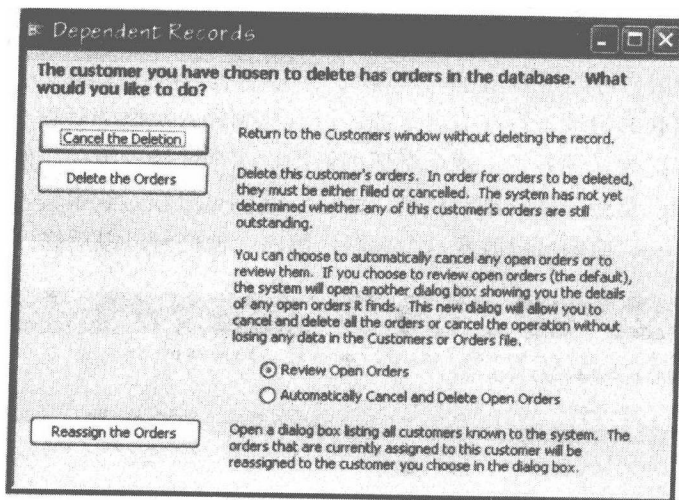


图19-3 该对话框解释了每一个选项的含义并允许用户去选择

图19-3所示的对话框可以让用户取消删除、级联删除、或者重新指定该订购记录所关联的不同的客户记录。注意该对话框警告用户打开的订购记录不能被删除（这是一条业务规则），并且提供取消（本质上是越过规则）或回顾这些订购记录的选项。

你可能应当在有效对话框中显示打开的订购记录，或所有客户的订购信息。你的目的是给用户提供尽可能多的信息，让他们能够做一个全面的决定。

19.3 业务约束

在上一节，我们探讨了内在约束，它是用来维护数据库结构和完整性的。如果用户输入

的数据与内在约束相冲突，你几乎无能为力。数据至少最终必须遵循关系模型的要求。但是，数据与业务约束相冲突是完全不同的情况。

正如我已说过的，你是从问题域派生出业务约束的，而不是关系模型本身。记住一个数据模型仅仅是一个简化的现实世界的某个部分的模型。作为系统的设计者，你需要尽你所能在模型中提取问题域的所有相关方面，但是尽管你尽全力，也不会总是成功的。即使你的模型在开始实施时各个方面都是完美的，但业务情况会发生改变。为了更加成功，你的系统必须能够很好地处理意外情况。

有两个原因用户可能会试图输入系统不准备处理的数据：一是他们偶然输入的数据，第二是的确与系统模型不匹配。（事实上，如果你算上故意破坏，将有三个原因，但是一般的系统分析者通常抱着悲观的态度，他们认为，破坏是极端稀少的情况，并且任何情况下，业务规则都不是处理它的最好方式。）

19.3.1 偶然输入

如果用户偶然输入了不期望出现的数据，你唯一能做的就是使他们尽可能简单地修正这个问题。至少，你应该解释这个问题到底是什么以及应当如何处理它。可能的情况下，你还应该提供可以让系统直接执行的更正方案。例如图19-4所示的对话框，当用户在一个日期字段中把月和日颠倒的时候，它就会被触发。该对话框对用户想要输入的日期做了一个合理的猜测并允许用户通过单击来选择更正后的日期值。

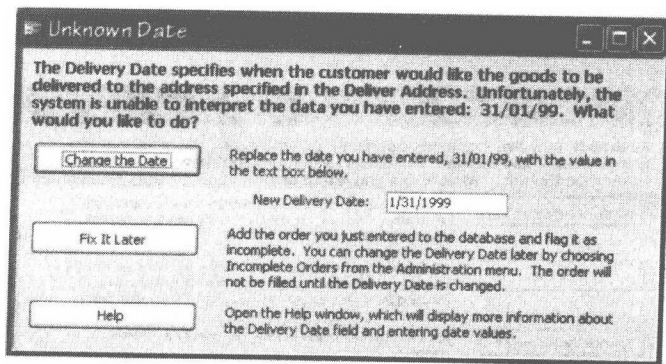


图19-4 该对话框解释了一个输入错误而不是发布最后通牒

当用户偶然输入数据时，它可能就是一个真正的意外，又或者可能是用户没有完全理解需要输入的内容。在图19-4中的对话框显示了一些解释DeliveryDate字段的用途的方式，但它也包含一个Help按钮来允许用户查找更详细的信息。我们将在第21章讨论辅助用户的细节。

19.3.2 现实与系统模型的对比

用户输入意外信息的第二个原因是现实与系统模型不太匹配。例如，用户可能试图在一份订单已经出货时修改该订单的某个信息，但是由于某种原因系统不能接受输入的信息。因为这么一来，出货日期（DeliveryDate）将会早于订购日期（OrderDate），这会破坏业务规则。如果不给用户提供某些指导，他们可能会输入任意系统可以接受的无意义的出货日期，那么数据库的完整性将受到损害。因此，有必要创造性地考虑尽可能多的意外情况，并且始终如

一地帮助用户解决它们。图19-5显示了一种可能的响应用户输入的DeliveryDate比OrderDate早的情况。这个例子是假定OrderDate是由当前系统默认的日子来设置并且用户通常不能编辑该字段。

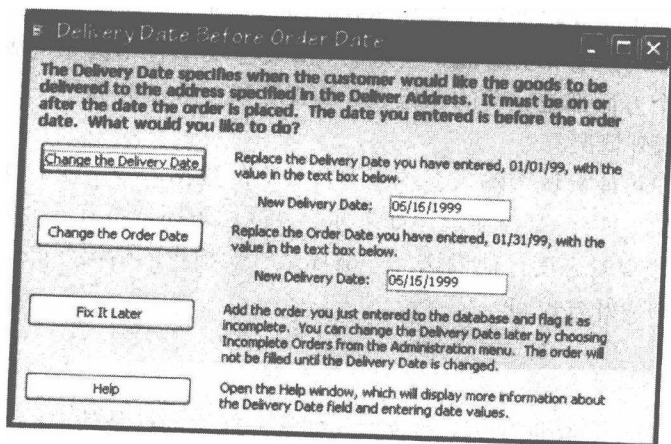


图19-5 该对话框允许用户回溯一个订购

并不是所有的业务规则都可能（或应该）被简单地越过。某些业务规则是不可能被破坏的，因为该情形是不可能的，或者是法律规定等。但是有时候问题不是“是否”一条规则被破坏了，而是“由谁”破坏的。例如，在图19-6中，一位用户试图输入直属某位经理的第六个雇员的相关信息，这就违反了一条规则：一位经理最多只能有五位雇员。

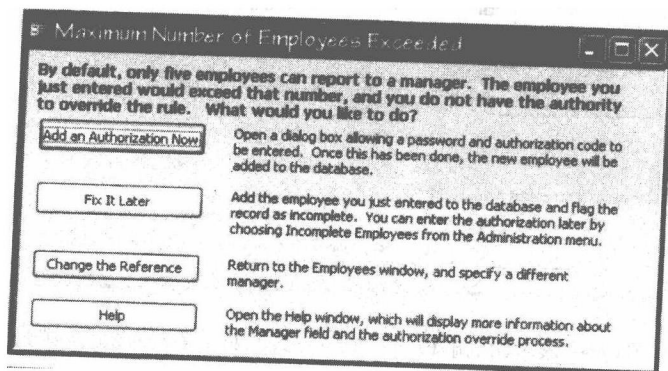


图19-6 该对话框表明只有某些被赋予权限的用户才能打破业务规则

当前用户没有权利来打破这条规则。但是系统提供了一个次级窗体来允许某个拥有权限的用户输入密码——假定是一位领导者或者一位经理。因为这种个别的权限并不是每个人都有的，所以第一个对话框也允许用户保存当前记录并提示之后需要输入权限密码。

提供打破业务规则的功能能够显著增加系统的可用性，但这也会给数据模型增加复杂性。在这个例子中，你必须在Employee表中增添一个字段来包含权限代码，并且在引用该权限代码时提供完整性约束。如果该权限被延迟，那么还必须提供一种机制让记录暂时失效。

一种方法是在Employee表中增添一个布尔型的字段来表明该记录是否有效。当用户已经解决了这个难题，或者在过滤记录以便生成报表时，这个标志可以被用来查找无效的记录。

有时候，它对明确该记录为何无效十分有用。并且在这种情况下，能够使用一个编码字段——例如，“MA”表示“awaiting Manager approval（等待经理批准）”或者“CC”代表“Credit Check Outstanding（未完成的信用检查）”。第二种方法在处理记录时提供了更大的灵活性。

需要对暂时失效的保存记录进行修改是显而易见的，并且它们大部分都是如此，但它们可以通过系统以意料不到的方式传播开。例如，暂时失效的记录应当被包含在报表中吗？一个临时的编码在决定如何处理这些报表时会十分有用，因为它能允许你决定哪些记录应当被排除在某个特定的报表或查询之外。

如果暂时失效的记录被包括在记录或查询中，则必须考虑它们是否应当被某种方式标记成不可靠的，比如通过将它们分组到一起并安排在一个单独的标题之下。

如果你已经决定暂时失效哪些记录，或者某个特定类型的暂时失效记录不应当包含在一个报表中，那么你必须接着决定是否与之关联的任何记录也应当被排除在外。举个例子，如果问题是那条关于一位尚未授权的第六位雇员的记录，那么是应当只把该雇员的记录排除在外，还是应该将所有的雇员记录都报告给同一个经理？

所有这些问题都必须在你决定是否提供打破规则的功能之前考虑。该功能的确提升了系统的可用性，但是无疑增加了复杂度。在简单的系统中，最好是拒绝那些不符合业务规则的记录，并且在计算机系统之外来处理这些意外情况。如果你决定在系统外处理意外情况，那么请确保在一个数据有效性对话框中说明这一点，不要让用户去猜测该如何进行下去。

19.4 小结

在这一章，我公然站在非正统的位置说明了数据库系统不必要强制实施数据库完整性约束。我们讨论了两种不同类别的约束：内在约束——管理数据的物理结构，由关系模型派生而来；业务约束——由问题域派生而来。

我们已经看到，允许用户违反内在约束是十分危险的，但是让违反业务约束的记录暂时失效而稍后解决它们是有意义的。我们还讨论了如何处理暂时失效记录的几个例子。

在下一章，我们将关注关于在系统维护中发布数据的不同方面的内容。

第20章 报 表

只有当数据库系统中保存的事实以某种有意义的方式组合起来时才能称之为信息，在这之前，它们只是些琐事。在这一章，我们将探讨将这些有意义的事实组合——即信息——提供给用户的有关问题。

注意：当我们在这一章中使用术语“报表 (reporting)”时，并不单纯地指生成打印的报表。我们是以更广泛的方式来使用该词，意思是建立在数据库中保存的数据基础之上的信息提供方式。这种信息可能是以打印报表的形式来提供，也可能在一个窗体中显示或作为数据集显示在数据表中。

过去，当计算机的成本远大于雇员的时候，计算时间是一个奇缺的资源，数据库系统在常规时间内只能产生少量的报表。如果想要系统作任何特别的工作，通常会无望而归。这样积压任务的问题在一般的MIS系统中存在了若干年，因此如果你需要一份报表，比如仅仅是要按客户来排序并按州统计合计，那么你把它交给你的秘书，让她用打字机打印出来好了。

如今，计算机很普及并且相当便宜，而秘书则成为较稀少的资源。现在用户必须能够告诉计算机系统，“我想要一个那样的报表，但是……”，这意味着你作为数据库设计者的工作将变得较为困难了。但是，它一定比听写速记要简单很多。（关于这一点，你完全可以相信我。）

在一个数据库系统中，提供报表功能的方法通常有两种：一是试图预知所有可能的报表；二是允许用户在他们需要的时候创建报表。大多数系统要求二者的结合。

那些能够在设计系统时预知的报表可以在实现过程中创建。我们称这种类型的报表为**标准报表**。你也可以在系统实现后提供给用户一些机制，让他们创建自己的报表。我们称这种类型的报表为**特殊报表**。我们将在本章讨论这两种类型的报表，但是我们首先要学习生成报表需要的数据排序、检索和过滤机制。

20.1 排序、检索和过滤数据

在写SQL语句时，排序、检索和过滤数据相对是比较直接的。你只需简单地在WHERE子句或ORDER BY子句中指定合适的条件即可。但是，当你想为用户提供这个功能时，还需要提供一个介于它们和SQL SELECT语句逻辑之间的过渡。

在这里困难的是SQL逻辑很难与一种自然语言来匹配。当一位销售经理需要在Wyoming和Florida公司的所有子公司列表时，她期望列表中同时包含在Wyoming的子公司和在Florida的子公司。但是SQL SELECT语句中的WHERE子句是：WHERE State = “Wyoming” OR State = “Florida”。从语言学的角度来说，正确的用法应当是“and”，但是在形式化的语法中正确的术语是“or”。这对用户来说是一个巨大的难题——SQL非常接近自然语言而导致它十分迷惑。

教用户如何书写SQL SELECT条件子句是可能做到的。事实上，我至少知道一位很受尊重的数据库设计者，据大家所说在这方面相当成功。我认为提供一个更直观的界面会使你自己、文档书写员以及（最重要的）用户避免直接使用SQL语法的麻烦。

幸运的是，你不是孤军作战的。Microsoft Access提供了构建SQL条件子句的界面示例。虽然将这些用户界面机制直接拷贝到自己的系统中并不总是合适，但这些机制至少提供了一个开端，这就是我将怎样在这里处理它。但Microsoft Visual Studio界面不支持任何这种机制，所以你必须花点功夫，使用Visual Basic代码来实现它们。

20.1.1 排序数据

Access提供了一个很好的排序界面，它使用Sort Ascending和Sort Descending命令。用户在他们需要排序数据的时候点击相应的控件，然后在记录菜单或者工具栏中选择合适的命令。很难想到比这更简单的界面了。

20.1.2 通过选择过滤

需要过滤数据时，Access提供的最简单的界面是通过Filter By Selection命令，以及与之对应的另一个命令Filter Excluding Selection。这两个命令工作的界面非常类似Sort Ascending和Sort Descending。用户在一个窗体中选择一个字段的内容，或者在该字段中放置插入点，然后选择Filter By Selection命令，Access就会过滤窗体中的记录集，而使其仅保留那些含有与所选字段相匹配的值的记录。换句话说，SQL WHERE条件是<filename> = <control value> (字段名=控件值)。

如果用户只选择了一个字段内容的开头部分，Access会将记录集限制在那些包含字段值以选定的字符作为开头的记录。例如，如果用户选择产品名称为“Chartreuse verte”的头三个字符，与之等价的SQL WHERE子句将是：WHERE [Product Name] LIKE “Cha*”。在Northwind样板数据库中，这样的查询返回包含产品名为Chartreuse verte、Chai以及Chang的记录。

如果用户在该字段中选择任何其他字符，Access将会返回所有包含该选择的记录，不论选择的字符在字段的哪个位置。比如，在前面的例子中，如果用户选择“ar”，与之等价的SQL WHERE子句是：WHERE [Product Name] LIKE “*ar”。在Northwind样板数据库中，这将返回10条记录，如图20-1所示。

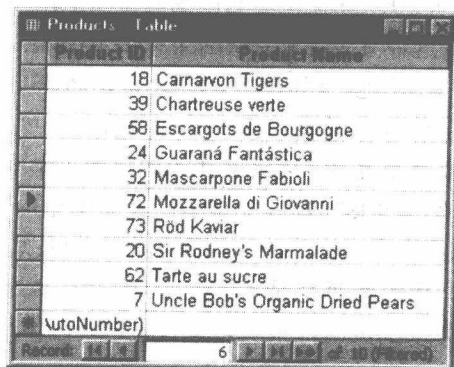


图20-1 如果用户在窗体或数据表单的Product Name字段中选“ar”，然后选择Filter By Selection命令，则将返回这些记录

注意：当用户选择Filter By Selection命令时，不要假设Access实际上是在使用SQL SELECT语句。当然，我认为那是实际的过程，但是我不能保证。因为无从得知那些向导在开发过程中等同于什么。

20.1.3 通过窗体过滤

Filter By Selection是简单的，它优秀的界面非常便于用户学习，但是它限制用户在单个字段上过滤。用户可以使用Filter By Selection命令来进一步缩小数据集的范围，但这将变得

十分繁琐。为了满足那些想要更强大的过滤界面的用户，Access提供了Filter By Form命令。图20-2显示了Northwind样板数据库中的Customers窗体，它是在用户已经从“记录”菜单中选择了Filter By Form命令之后的形式。

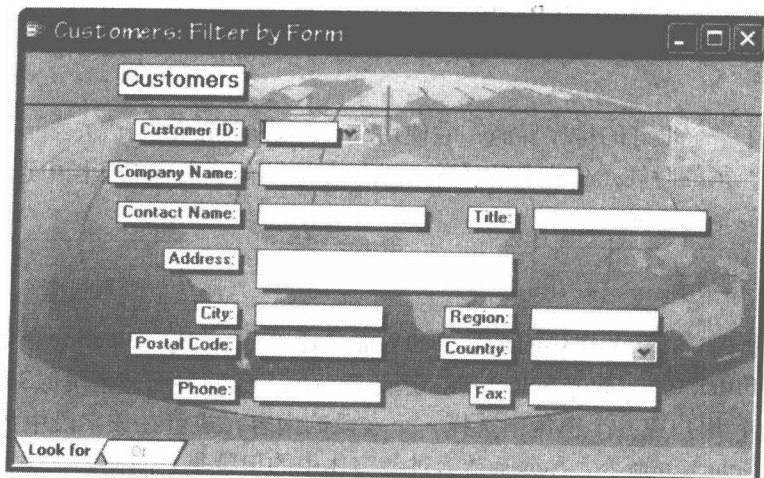


图20-2 Filter By Form命令允许用户在多个字段中过滤信息

在一个Filter By Form窗口中，用户可以在窗口的多个选项卡中为多个字段设置过滤值。那些在Look For选项卡中的过滤值将以逻辑AND来组合，而那些在Or选项卡上的值将以逻辑OR来组合。这种方法没有完全消除AND及OR在语言学和正式用法之间的差异，但是它已经十分接近了。

默认地，Filter By Form窗口将来自源窗体的每一个文本框作为一个组合框来显示，这个组合框包含了该字段的所有当前值。可以通过设置文本框控件的Filter Lookup属性为Never来取消这个功能。如果Filter By Form窗口显示组合框，那么将限制记录必须完全匹配用户选择的值。如果在窗口中显示的是文本框——这意味着设置Filter Lookup为Never——那么用户可以输入一个必须完全匹配的值，也可以输入一个表达式，比如LIKE “CHA” 或IS NOT NULL等。需要根据记录集的大小（不能要求用户等待Access往组合框中填充100 000个项）以及用户要求的灵活性来决定是否在Filter By Form窗口中使用组合框。

20.1.4 高级过滤和排序

Filter By Form方式，或者基于它的一个界面已经足够来解决大部分问题。但是，如果你的用户对Access的查询设计非常熟悉，或者你正在使用Filter Lookup属性来允许用户从Filter By Form窗口的一个组合框中选择值，并且还希望他们能够输入过滤表达式，那么，Access的Advanced Filter/Sort（高级过滤和排序）窗口将会十分有用。如图20-3所示。

Advanced Filter/Sort窗口为查询提供了Design View所提供的功能的一个子集：它仅控制SELECT语句中的WHERE和ORDER BY子句，它不提供通过改变字段列表或连接其他记录集而修改所返回的记录集的基本结构的功能。

Advanced Filter/Sort窗口是一个过滤界面，我在Visual Basic中不想再重复这些内容了。我想它肯定也能完成这些功能，但这的确不是个简单的工作。如果你需要该功能，那么你可以就选择Access作为你的开发工具或者寻找一个第三方产品插入到你的系统中。

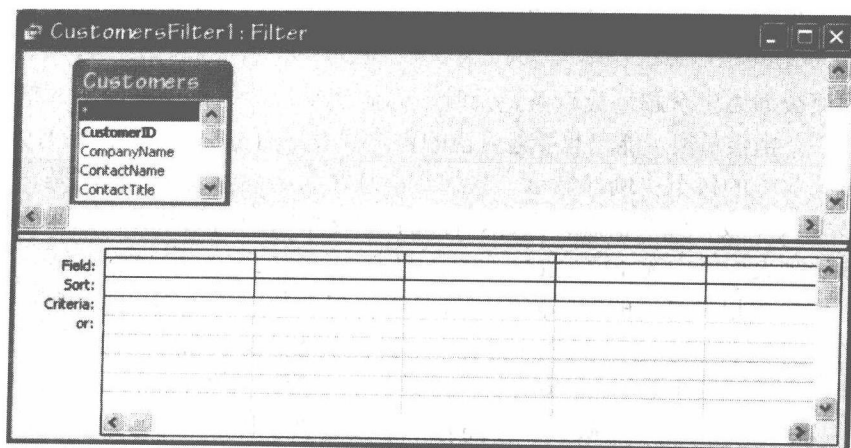


图20-3 如果用户在Northwind样板数据库中打开的Customers窗体中选择Advanced Filter/Sort, Microsoft Access 2000将显示这个窗口

20.1.5 微软自然语言查询

如果选择SQL Server作为数据库引擎,那么可以考虑实施微软自然语言查询(Microsoft English Query)。自然语言查询不仅仅是一个排序和过滤界面,它还为数据库提供了完全的自然语言界面。

为了在应用程序中实现自然语言查询,需要通过创建一个被自然语言查询称为的“应用文件(application file)”,来将问题域的语言映射到数据库模式上。创建一个自然语言查询应用文件并不难,但是却十分繁琐。同一个好的帮助文件索引一样,它要求花很多的时间来预测用户将要使用的词汇,并将它们与模式中的实体和属性关联起来。

一旦创建好了该应用文件,将自然语言查询集成到数据库应用程序中就十分容易了。数据库应用程序只需简单地给自然语言查询引擎提交一个用户的自然语言问题,然后接收返回的一个SQL语句。当然,理论上是这么容易,但实际上,你的应用程序很可能接收到一个错误消息,表明该引擎无法理解用户(该用户杜撰了一些新词)提出的问题的表达方式。

在合适的环境下,自然语言查询是一个十分强大的工具。如果你有一个复杂的数据库模式,并且有很多用户在做查询,那么自然语言查询所提供的自然语言界面会是一个极好的解决方案。

20.2 生成标准报表

几乎每个数据库系统都拥有一定数量的可以提前定义的报表。在分析工作过程中你可以发掘大部分的这类标准报表,但是依然十分值得去花点时间和用户仔细检查一下数据库模式,从而考虑是否还有其他对他们更有用的报表。

20.2.1 清单报表和明细报表

对于系统中的每个实体,考虑使用清单报表和明细报表。清单报表就是列出了实体中的每个实例(也就是每个表中的记录)的报表。有时候可以简单地以字母顺序来排列这些清单

项。更多时候,你应当以某种方式来将它们分组。例如,客户可能按州、地区或者销售员来分组。

当报表的原数据表包含超过数百条记录时,你应该给用户仅提供打印所选范围的记录的选项。例如,一个销售员很可能只想要他自己的客户列表,而不是系统中的所有客户。

一张清单为实体的每个实例都显示一些细节信息,而明细报表显示的则是一个指定实体的所有(至少是大多数)细节信息。同样,你通常也想给用户仅提供某种方式让他们选择打印的记录。多选列表框对于选择记录是一种好的机制,因为它不要求连续选择。

但是,一定要牢记列表框的实际限制。如果表格包含数以千计的记录,则必须使用一组列表框来允许用户逐步缩小记录的范围。此外,你可以选择使用其他类型的控件。例如,你可以考虑使用一个功能上类似于Microsoft Word的打印对话框的文本控件,在其中指定要打印的范围。将一个范围内的记录用破折号分隔开,或者用逗号将单个记录隔开,这并不是什么难事。

20.2.2 总结报表

更有用的报表是我认为的“切片和切块”报表:以不同方式组合和比较的总结数据的报表。相对清单和明细报表来说,总结报表要较难实现一点。按区域或销售员划分的销售百分比、购买每种类型的产品的客户数量等都是这种类型的报表。

总结报表可以很好地图形化表示,并且Microsoft Graph和多种第三方工具使图形化报表的实现十分简易直观。但是,我建议你將图形作为文本数据的附件,而不是它的替代。一个基于文本的销售总结报表可能查看起来不那么方便,但是它可以被输出到统计分析工具或者电子数据表中,例如Microsoft Excel,以实现进一步的操作。

20.2.3 基于窗体的报表

除了从数据库模式派生的报表之外,系统中的窗体也可以考虑作为有用报表的一个来源。我在数据库系统中提供了打印主要窗体的功能,这可以作为某种课程。它们很容易实现,并且有益于用户检查他们的工作或者做一份快速的打印输出给其他人看。

有时候,一个实体细节的报表包含作为一个窗体的相同信息,并且你可以使用这个细节报表而不需要创建一个新的窗体报表。但是,大多数的情况是每个实体细节报表包含额外信息或者与窗体报表的格式有所不同。基于窗体的报表十分廉价和简单,因此我通常会给我的用户同时提供细节和窗体报表。当用户点击工具栏上的打印按钮时,基于窗体的报表是默认的打印报表,而细节报表是通过菜单调用的(也有可能来自工具栏)。

20.2.4 报告界面

在用户界面中提供制作报表的功能并不困难。同任何命令一样,可以有三种方法来使该命令对用户可用:通过在窗体上添加一个菜单、工具栏按钮或者一个命令按钮。

当然,打印一张报表的命令必须让用户清楚是哪张报表要打印。通过菜单就很容易实现:只需要使用报表的名称作为“报表”菜单项就可以了。可以使用报表名称作为工具栏按钮的“工具提示”或者一个命令按钮的标题,但是应当在该名称前加上动词“打印”。例如,“客户列表”就足够作为“报表”菜单上的一个菜单项,但是相应的“工具提示”或者命令按钮的

标题应当为“打印客户列表”。除了可以和Windows的指南相一致外，这样的文本可以让用户清楚系统将要打印一张报表，而不是打开一个包含报表的窗口。

除了确定将在哪里给用户提供系统报表（通过菜单、工具栏或者命令按钮）之外，还必须考虑如何提供。数据库系统通常会包含数十，甚至数百张报表，将它们罗列在一个庞大的多栏菜单中显然是没有任何意义的。幸运的是，限制那些对用户的当前事务有意义的报表还是很容易做到的。比如，一位用户不可能在输入销售订单的过程中想要打印一列雇员的电话号码。

如果认为让所有的系统报表立刻可用是比较合适的，那么可以提供一个对话框来显示报表的类别，并允许用户选择他们想要打印的报表。这种方法对于用户需要一次打印多张报表也会十分有用。通过允许他们在报表对话框中选择一次打印的报表数目，可使他们只需要点击一次打印按钮后就做自己的事儿了。

对于那些经常打印的报表，应当作为一个集合——比如月底总结——我以各种不同的方式使用这样的技术。我在对话框中列出所有经常打印的报表，将它们作为一个集合，并且默认地选中它们。在将整个集合送往打印机之前，用户可以随意添加任何只是偶然打印的相关报表，或者去掉某个标准报表。

与通过一个菜单项批量处理相比，在一个集合中列出每个报表会给用户增加一点负担，但是我认为这种额外的灵活性值得用户多做一次鼠标点击。一批中的某张报表可能需要重新打印，也许是由于打印机的错误或者某人在上面洒了点咖啡。因此，该对话框应当既提供将这些报表集合组合起来打印的功能，但也允许单独打印每张报表，这样可以消除在一个菜单上列出每个报表的需要，还可以避免只因为一份报表有问题而重新打印整个报表集合。

20.2.5 处理打印机错误

你的系统必须能够处理打印机的问题或者打印输出的问题，并且由于这一点，某些常见打印情况可能就很难处理。例如，某位用户可能想要打印所有尚未打印的发票。这是一个很平常的要求，但是由于系统不能得知任意给定的报表是否已经被打印过了，这个问题对于系统来说则会很难处理好。系统只清楚它把报表送向了打印缓冲区，而这是完全不同的两件事。

一些设计者通过在将报表送到打印机之后显示一个消息框，并要求确认报表打印成功，以此来处理可能的打印错误。这种方法有一定作用，但是它要求用户在打印输出之前不能使用系统。如果打印任务偶尔在打印机的任务队列中排在某人1000页的手稿之后，那这位用户可要等上好一阵子了。进一步来说，大多数的报表在第一次都能正确打印，因此这种拖延99%是没有必要的。

我倾向于在打印错误可能出现的时候解决它们。回到我们之前的那个例子，如果系统只需要打印未打印的发票，那么你只要相应的表中添加一个字段。如果系统只要求用户确认那些发票是否打印成功，那么只要一个Yes/No或者一个布尔型字段就可以了。但是保存一个日期或者打印任务编码会更容易一些。然后就可以添加一个命令来允许用户在一个打印任务或者一张发票上记录打印问题。

如果报表的打印在一天内不会超过一次，那么就可以用当前日期作为一个标记。但是，更安全的做法是为每一个打印任务生成并保存一个唯一的编号。如果发生了任何错误，用户只需要选择相应的打印任务并且系统可以将包含该打印任务编号的字段置为Null。包含Null的

记录将自动包括在下一个打印任务中。另外，系统还可以显示那些包含在错误打印任务中的所有记录，并且允许用户只选择特定的记录重新打印。

用户如何识别一个特殊的打印任务呢？可以在报表的页脚中包含打印任务编号，但是我倾向于设定一个系统表来保存报表的名称、打印任务编号、报表打印的日期以及（如果存在）实施该任务的用户名称。之后就可以给用户一份打印任务列表的描述信息，而不是让他们记住一个无意义的编号。选择“这些发票是我在周三早上打印的”是十分容易的。

有时候还有比某个记录是否包含在下一打印队列中更麻烦的事情，它们和工作过程的一部分。例如，会计系统有时候会把生成报表作为月末处理的一部分。一旦该月过去，某些特定的数值将被重新初始化，因此就没有办法（或者至少没有简单的办法）重新生成报表。我认为这是一个非常糟糕的设计策略，但是它很常见。

由于打印的不可靠性，我努力将报表生成和记录更新（不是“已打印记录”更新）完全作为单独的任务，并且我建议你也这么做。如果该工作过程的确要求更新记录必须与打印报表连接起来，那么最安全的方法是将进一步的系统处理推迟到打印报表的人确认报表打印成功之后。

确认一个成功的打印任务可能就像做一次后台处理，因此你并不需要让系统等到该任务被确认。比如，你可以在状态栏上放置一个图标，当点击它的时候，显示确认对话框并且完成表的更新。一定要给用户显示一些信息来说明他们需要做什么。

20.2.6 自动和随选打印

在生成标准报表中另一项需要考虑的事宜就是它们是否应当随选打印，或者自动打印，或者两样同时需要。作为一般规则，我们让所有的报表都可随选打印，唯一的例外情况是如果报表显然是作为工作过程的一部分，比如在用户输入一份销售订单后生成一张发票。自动报表功能也应当根据需要提供，这样可以处理那些讨厌的打印问题。

请注意，那些绑定在某个工作过程上的报表，比如发票，可能一次打印一张或者一批。例如，你可以一次打印一张发票来对应输入的销售订单，或者可以存下所有未打印的发票，在数据输入完毕后一次将它们全部打印出来。我已经成功地使用过这两种方法。如果用户有一台本地打印机，那么我倾向于一次打印一张发票；如果他们使用网络打印机，我会选择批量打印。你应该根据用户的情况来选择——打印机设置服从变动。

一些设计者选择让系统自动生成定期打印的报表，例如每周或者月末的报表，但是我喜欢将这些也作为随选。让系统自动生成报表是项十分复杂的任务。系统必须计算何时该报表需要打印（允许周末和节假日），并且跟踪该报表是否在正确的日期打印了。如果多人使用该系统，它还要确定哪位用户或者哪一类用户可以触发打印任务，以及如果那一类用户没有按要求在日期登陆系统又该如何处理。

和所有与自动生成报表相关的问题相比，让用户在他们方便的时候从菜单中选择“Print Weekly Reports”就显得容易多了。不管怎样，你需要提供这样的菜单项（或者一个报表问题对话框），因为用户需要在发生打印问题的时候重新生成报表。

用户不太可能会忘记打印每周、每月或者每季度的报表，但是允许用户指定报表运行的周期会更好一些，以免他们会忘记。通过使用当前时间周期作为默认值并允许用户改变这个值，可以使他们较容易地修正任何疏忽。这种技术还允许那些定期打印的报表能够在当前时

间周期结束前打印。如果还有时间做一些超出成本预算的事情，那么能够针对预算检查一下性能会显得十分有用。

20.3 生成特殊报表

一个数据库系统将要支持的工作过程有时候都已经完全定义好了，因此你能够提前指定所有系统必须生成的报表。但是，更通常的情况是需要给用户提供一些让他们能自己来配置或设计报表的灵活方式。

至于需要有多灵活则应当由用户的需求来决定，并且每个系统的要求都不一样。灵活程度的范围可以从提供用户完全设计报表的能力到简单地允许用户为预定义的报表指定附加的筛选条件。

20.3.1 报表设计器

给用户提供一个报表设计工具对实现来说是个简单的选项，如果你可以使用某个商业的工具，比如Microsoft Access或者一个第三方报表设计器，比如Crystal Reports。

报表设计器可以提供给用户无限的自由空间，让他们设计出所需的报表。但不幸的是，这种灵活性是有代价的。不过并不是报表工具本身的成本很高，但是如果你给数百名用户提供他们根本不需要的Microsoft Access的完全拷贝，这种基本成本就非常重要了。

更多的成本在于必须培训这些用户使用复杂的工具来设计报表。用户不仅需要知道如何使用报表设计器来规划报表，并且他们还必须对数据库模式有一个肤浅的认识，因为只有通过数据库模式才能访问他们需要的具体数据。所有这些都需要给予用户充分的时间和培训。但是这些人都已经有工作了，而构建定制的报表并不是他们的工作。

20.3.2 自定义的报表设计

为了让报表设计过程对用户更有效，实现一个自定义的报表设计功能通常比使用一个商业的工具更合适。理论上，提供一个同Microsoft Access报表设计器一样具有一定的灵活性并且可以自定义支持系统的数据库模式的报表设计器是可能的。但是，这样的工具一定十分昂贵，并且很难想像这种情况下的成本会是合理的。更普遍的解决方案是提供一套预定义的报表界面并且允许用户指定将要显示的数据。

Access报表向导可能与自定义报表的实现模型十分接近，但是它也是提供预定义窗体布局的一种有用方法的一个例子。报表向导提供了广泛的格式化功能，它通过允许用户从一个预定义报表界面和风格的列表中进行选择的方法避免了报表设计的复杂性。一旦报表向导生成了报表，用户就可以进一步使用Access界面操作它了。

允许用户分别指定报表的布局 and 风格给用户提供了对报表的控制。或者，你可以将布局 and 风格以不同的方式组合起来以简化这个过程。大多数用户要求的“自定义”报表实际上仅仅是标准报表的变形而已。从另一个角度来说，他们所需要的只是“就像这样，不过……”的报表，并且“不过……”后面的内容就是排序或者筛选条件。

在我自己的工作中，我使用的分类方法与Access报表向导差异很大。我将一个特殊的报表分为两个部分，分别称之为“格式”和“条件”。格式是将报表的布局 and 风格组件组合在一起，并且还指定了需要打印的字段（以及窗体基于的表或查询）。事实上，这是一个报表对象，

它还要在运行的时候根据用户指定的条件进行完善。

条件指定应用于格式上的排序和筛选。我一般会给用户提供一种保存和重用条件的方法。我们之后马上就会看到它是如何发挥作用的。允许用户定义分组级别有时候也很有用，但是一般来看，我发现没有太大必要。

由于用户只需要指定两个方面，因此，我使用一个对话框来产生自定义报表，该对话框的一般结构如图20-4所示。（如果你对特殊报表采用这种方法，那么你应该修改这个模板来与系统的用户界面风格保持一致。）

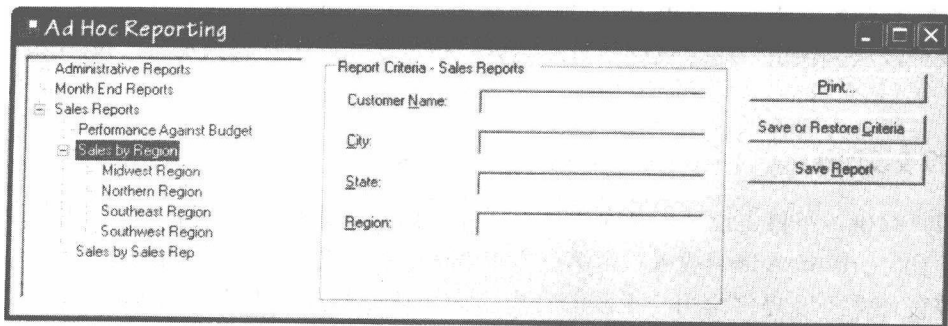


图20-4 使用这种基本的结构给用户特殊的报表

该对话框被分为三部分。在左边的面板中，一个TreeView控件给用户展示了可供选择的格式列表。不一定要使用TreeView控件，也可以使用一个列表框甚至是一组单选按钮。在这个例子中，这些格式被分成若干类别。如果你有很多格式的话，这种方法是十分有用的。例如，将这些格式分为“Administrative Reports”、“Month-End Reports”以及“Sales Reports”，这样会使用户更容易找到他们想要的报表类型。

图20-4中层次的最底层是“reports”。在这种情况下，一张报表就是一个已经保存的条件和格式的组合。例如，在一个销售报表的条件中指定了Region为“Southwest”，那么该用户就可以保存一张名为“Southwest Region Sales”的报表。如果你有更复杂的条件说明，那么这样的功能可以为用户节省很多时间。

此对话框的中间部分的面板允许用户指定排序和过滤条件。有时，可能在条件面板中为所有的格式使用一个控件集，而对那些对用户选择的格式用不上的控件有必要设置为禁用状态。有时对于建立条件来说，每个格式可能需要完全不同的控件集。在我的工作中，我倾向于折衷的方案，正如图20-4所示的，我将报表格式划分为类，并且用类别决定条件控件。

图20-4的对话框中的右边面板包含一组命令按钮。Print按钮将显示一个子对话框，它允许用户设置打印选项，比如打印份数或者打印机。如果这些选项不可用，那么最好能够提供两个命令按钮，Print和Print Preview，这样就不用给用户显示额外的对话框了。

Save Or Restore Criteria命令按钮将显示一个对话框，它与图20-5所示的类似。这个简单的对话框显示的条件与在主定制报表格式的中间面板部分显示的相同。用户已经保存的条件显示在左边面板中的列表框中。在中间面板的条件框列表选择一个条件。Save As...按钮首先让用户提供一个文件名然后保存指定的条件，而Restore按钮会将条件加载到主窗体上。

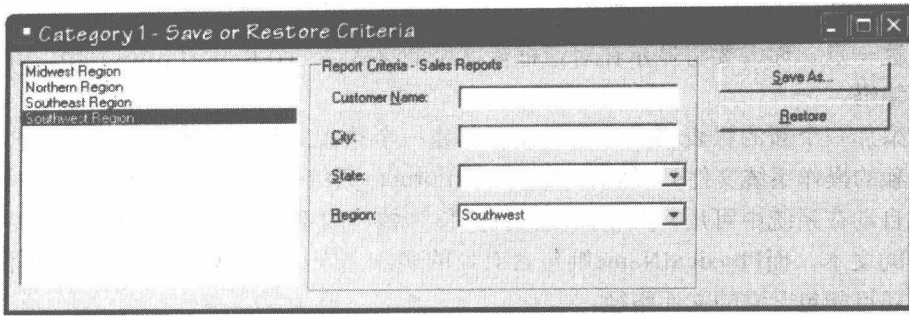


图20-5 该对话框可以从主自定义报表窗体上调用，它允许用户保存和恢复报表条件

以这种方式来保存条件的功能实现起来相当简单。你只需要为每个类别准备一张表，其字段是面板上的每个控件的名称。在多用户情况下，你需要决定是否要将已经保存的条件开放给所有用户，或是允许每个用户维护他们自己的集合。如果条件是共享的，那么这些表必须和其他共享的数据一起保存在主数据库中。如果每个用户维护他自己的条件集合，那么这些表应当保存在本地的前端数据库中（或者本地数据库中，如果你的应用程序是用Microsoft Access之外的工具编写的）。

这两种技术并不是互相排斥的。你可以通过在条件表中包含用户的名字或者通过显示共享和本地数据表的合集，很容易同时提供共享的以及用户指定的条件。我通常会在共享表格中包含用户的名字，因为它可以很容易地支持“漫游”的用户，他们可能使用多台电脑来访问系统。

保存和恢复条件的功能允许用户在一个类别的基础上保存条件。因此该条件可以被应用于任何属于该类别的报表（假定这些报表都共享条件说明）。

但是有时候，将条件和一个具体的报表格式连接起来会更合适一些，这就是图20-4中最后一个按钮的作用。同样，它实现起来也很容易。你只需要为每一类条件准备一些表（可以重用那些用于保存条件的表，如果你已经实现了这项功能的话），还需一张表将报表格式和条件连接起来。这种结构如图20-6所示。

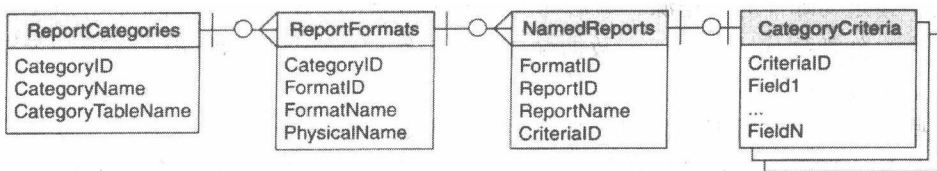


图20-6 该表结构允许用户保存条件和报表，还能允许在系统运行的过程中添加报表

ReportCategories表包含一个叫做CategoryTableName的字段，它允许系统标识包含该格式类别特定的条件表。如果所有的报表都有相同的条件结构，那么该字段就不是必要的。另一方面，如果所有报表格式都有不同的条件结构，那么在ReportFormats表中就应该有该字段。

图20-6中显示的表结构还允许在运行的时候配置系统中的报表，这就是在ReportFormats表中的FormatName和PhysicalName字段的含义。如果你的报表格式是单独定义的，或者是作为数据库中的对象（就像Access中的报表一样）定义的，又或者是作为单独的文件（和很多

第三方报表工具一样)定义的,那么你都可以使用间接的方式在任何时候向系统中添加报表。为了做到这一点,你需要将显示在对话框窗体中的格式罗列在ReporFroms表中,而不是对列表实行硬编码。

为了添加一个新的报表格式,你只需要创建一个格式对象(在Access中是一个报表或者是一个单独的操作系统文件),然后再在ReportFormats系统表中添加一条记录。这样这个新的格式就能自动在系统中可用,而不需要任何人接触核心的系统功能。FromatName字段包含显示给用户的文本,而PhysicalName则包含对象的实际名字,并且如果这些对象是保存在外部的,那么还可能包含它的文件路径。

这里描述的自定义报表的方法对于用户来说虽然比一个全开放的报表设计工具简单,但是它对一些应用程序也存在危害。你的用户所需要做的就是偶尔为一些预定义的标准报表指定额外筛选条件。你可以通过添加一些按钮来经常调整这些简单的条件,它们把条件指定给一个自定义的打印对话框,而不是实现一个之前描述的完全自定义的报表用户界面。

20.3.3 标准信件

一种特殊的报表就是标准信件。有时候,一封标准信件的内容是固定的,但是更多的时候,用户需要从样本文件的段落中选择,然后让系统将它们组合起来构成一封信。不论信的文本内容是否固定,我的观点是数据库报表不是一个生成信件的最好方法。

虽然数据库报表格式化的能力越来越强大了,但是它们不能与专门为这项任务设计的文字处理器的格式能力相比。另外,多数用户都希望能够在信件被打印前住其中添加文本,但是允许他们对数据库的报表这么操作是不明智的,因为任何对标准结构的修改都会成为永久的。

总而言之,应该尽可能地使用数据库来维护名字和地址,甚至保存样板文件的段落。但应当将这些数据输出到文字处理器比如Microsoft Word中来处理,这样可以有更多精巧的格式并让用户在打印它之前操作这些文本。

幸运的是,将这些标准信件传送到文字处理器越来越简单了。以前艰难地将“动态数据交换”(Dynamic Data Exchange, DDE)命令字符串送到一些不匹配的、结构糟糕的应用程序的日子已经过去了。现在比如在Microsoft Word中,你可以直接将一张Access表或者查询加上一份文档指定为邮件的数据源。之后你就可以使用Visual Basic for Application (VBA)来将文档和来自你数据库应用程序的数据混合起来,然后将结果显示给用户作进一步操作,或者直接传送给打印机。

在某些情况下,你可能需要跟踪那些由系统创建的标准信件。如果你不允许用户在打印前自定义信件,那么就没有必要在数据库中保存这些信件。你只需要保存信件传送这个事实,可能还有日期及发送人的名字。另一方面,如果用户通过合并样板文件的段落来创建信件,那么你可以使用一个复杂的实体来构建这个过程,如图20-7所示。

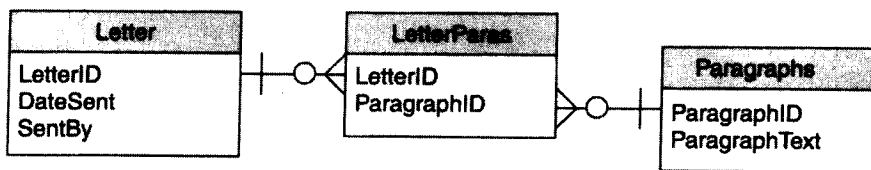


图20-7 这个结构可以保存包含在特定信件中的段落

如果系统允许用户在打印前定制信件，那么最好不要在数据库中保存这些信件的文本，因为Access和Microsoft SQL Server在处理大量文本方面都不够有效。更好的方法是只保存物理文档文件的位置和名称，但是这意味着系统必须准备处理文件的移动、重命名或者删除操作，这通常需要用户协助。

20.4 小结

在这一章，我们讨论了给用户提供的信息不同方面，这些信息都是基于数据库中存储的数据的。这通常意味着要生成打印报表，但是它也可能意味着通过在数据表中显示窗体或记录集的方式来提供信息。

我们一开始讨论了Microsoft Access提供的对数据进行排序和筛选的技术。即便应用程序不是用Access实现的，这些技术也提供了一些功能示例。

我们还探讨了系统可能提供的不同种类的标准报表，包括清单和详细报表、总结报表以及系统中基于窗体的报表。之后我们关注了如何将这些报表集成到系统的用户界面中去，以及一些涉及处理报表错误的问题。特殊报表的生成讨论得更详细些，并且展示了为了提供这项功能所使用的技术。最后，我们探讨了标准信件的生成，它通过将数据库的数据管理功能以及文字处理器的先进格式化能力综合起来实现。

在下一章，我们将转向用户帮助的主题，并且关注一下那些可以应用到数据库系统的用户界面中的用户帮助的形式。

第21章 用户帮助

在某种意义上，我们在本书第四部分讨论的所有内容都属于“用户帮助”的范畴。设计用户界面以及选择窗体结构和控件的目的都应当是帮助用户完成他们的任务。例如，在第19章中讨论的数据完整性问题就是在不妨碍用户的前提下帮助他们避免意外。在本章中，我们将更多地关注那些能够应用到用户界面中的更直接的用户帮助形式。

21.1 用户级别

我们已经讨论过共有三类人将会使用你的系统。初学者需要知道你的系统是做什么的，中级用户需要知道如何实施具体的任务，而专业用户想要知道如何快速的完成任务。对于不同级别的用户需要有不同类型的帮助方式。介绍性的界面和向导对于初学者十分有用，但却会妨碍中级用户，对于高级用户来讲，它们则成为一种干扰。

不仅需要考虑为不同类型用户提供不同的支持机制，而且需要考虑让这些不同机制共存的最佳方式。例如，在程序启动时为初学者显示的介绍性对话框应当包含一个“不再显示我”的复选框，这样使得更高级的用户可以略过该对话框。那些支持中级用户的工具提示和状态栏信息对更高级的用户不会造成太多麻烦，但是最好提供一种能够关掉工具提示和状态栏信息的机制，以防万一。我们将在本章后面详细讨论为不同类型的用户定制用户界面的方法。

除了支持机制的共存之外，还要考虑系统如何帮助用户从一个能力级别转到另一个级别。你应该依赖于系统的文档和外部培训，但是这些机制对于提高用户的能力都不是十分有效的，我们之后讨论其原因。一种更好的技术是在用户界面本身中帮助用户。

大多数较复杂的系统都会为每个命令提供多种途径。例如，可以通过从“File”菜单中选择“Save”来完成对记录更改的保存，也可以点击一个工具栏按钮，或者敲击Ctrl-S键。这每一种途径被称作一个**命令向量**，并且每个命令向量是适合于不同级别的用户。

初学者和中级用户倾向于依赖菜单来提示他们可用的功能；中级用户还会更好的使用工具栏按钮；而专业用户更多地使用快捷键。为了帮助用户从一个级别提升到另一个级别，需要在整个系统中不断为每个命令显示所有这些向量。

图21-1显示了Microsoft Access中默认的“File”菜单。请注意，“Save”项显示了该命令的所有向量。这个帮助记忆的访问键Alt-F-S通过为单词Save中的大写字母S添加下划线来表明，（“File”菜单中的F也加了下划线。）和Ctrl-S快捷键一样，在工具栏上用于该命令的图标也显示出来了。通过在菜单中显示所有的命令向量，该用户界面可以帮助用户学习实施任务的快速方式。

注意：Visual Studio.NET不允许在它自身的菜单上显示工具栏图标，虽然实现一个显示它们的OwnerDraw菜单并不十分困难。不过，有多种第三方控件能够支持在它们的菜单上显示图标。

Microsoft Access中的菜单创建模式允许包含图标，但是那些内置的图标太小了，因此你可能总是需要使用Button Editor来画自己的图标。Visual Studio和Access都很容易创建自动显示在菜单中的快捷键。

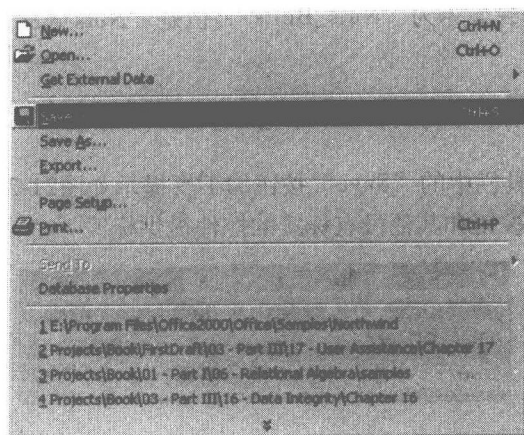


图21-1 Microsoft Access 2000中默认的“File”菜单显示了所有“Save”命令的向量

显示多个命令向量并不会有任何共存问题，因为它是一种被动的机制。所有的用户帮助机制能分为**被动机制**——组成用户界面的一部分，**反应机制**——在响应用户动作的时候调用，以及**主动机制**——试图预测用户的需求。我们在本章中将依次讨论这三种机制，并在最后大致讨论一下培训资料 and 用户自定义。

21.2 被动帮助机制

被动帮助机制包括所有嵌入到用户界面中指导用户完成任务的线索、指针和解释。与反应帮助机制不同的是，被动帮助机制不要求用户做任何事情——正如其名字一样。

控件标签、菜单名称，甚至窗体和对话框的标题都是被动帮助机制，因此你应当谨慎考虑如何对这些元素命名。一定要选择尽可能清晰的描述实施的动作或者输入的数据名称。

除了名称之外，还有很多其他的被动机制。我们在这里只讨论三种：帮助记忆的访问键、工具提示和状态栏。

21.2.1 帮助记忆的访问键

帮助记忆的访问键为用户同时提供了指引功能和被动帮助。因为它们为用户在系统中导航，因此访问键提供了指引功能。又由于它们总是在控件标签上添加相应的下划线，因此访问键还提供了被动用户帮助。任何对Microsoft Windows有一定经验的用户都会十分熟悉“Alt-下划线字母”的样式。

你应当为所有的菜单项和控件都提供帮助记忆的访问键。选择哪个字母作为访问键应当遵循下面的优先顺序：

1. 菜单项或者控件标签的首字母
2. 菜单项或者控件标签中与众不同的辅音字母
3. 菜单项或者控件标签中的元音字母

在Access和Visual Basic中定义访问键是很容易的：只需要将访问键字母放在标签中的前面，后面加一个“&”符号。在这两个环境下，系统将会在标签中给该字母加下划线并且为你处理导航功能。（为了在标签中显示“&”字符，而不是下划线字母，你需要使用两个“&”

符号。“Nuts & Bolts”将会显示为“Nuts_Bolts”，但是“Nuts && Bolts”将会显示为“Nuts & Bolts”。)

21.2.2 工具提示

Access Form View工具栏中的“Save”按钮的工具提示如图21-2所示。工具提示背后的原则很简单：它们为工具栏按钮和其他没有标签的控件提供标签。

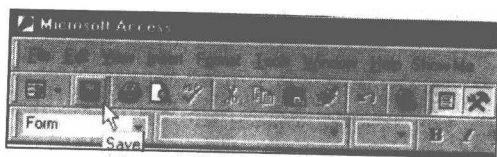


图21-2 工具提示表明了那些没有标签的控件的用途

并没有那么激动人心，对吗？不过，虽然工具提示是单调和简单的，但它们对系统可用性的影响是巨大的。如果你必须选择图标来表述系统的功能，那么你就知道让一个图标来传达信息会有多困难。选择一个“Save”的图标并不是太困难（至少现在不那么困难，因为我们已经都看过Microsoft Office中的磁盘图标），但是“Open Customer Form”按钮呢？你可以使用一个小图形，但是如果你还需要打开Employees以及Vendors窗体的图形呢？你的图标含义就可能有些含糊了。

幸运的是，工具提示减小了很多找寻一个能自我解释的图标的压力。一半的机率下，大多数人能够很好地将一个图像与一个含义联系起来——这就是记忆人名的原则——而工具提示给了他们另一半机率。

因此，你可以在Access中使用一个小鱼的图标来表示Customers。（Access的设计者脑子里在想什么呢？）在用户第一次看到你的工具栏上的这个小鱼的时候肯定不会把他们理解成Customers窗体，但是工具提示会解释该图标的含义。只要你在任何引用客户的地方——菜单、窗体以及文档中都使用该图像来加强该联系，那么用户适应起来就没有任何问题了。

在Access或者Visual Basic中实现工具提示，你只需要设置相应的属性。工具提示的文本必须很简短——最好是一到两个单词。记住，该工具提示功能是作为标签使用的：它的目的是提示用户该控件是干什么的，而不是教用户如何使用它。

如果该控件复制了某个菜单项，那么应当在工具提示中使用和菜单相同的单词或短语。如果该控件不是复制的菜单项，那么使用一个最能描述该控件功能并且能与其他控件区分开的名词或者动词。例如，如果一个工具栏包含三个按钮，分别打开Customers窗体、Suppliers窗体和Employee窗体，那么可以分别使用工具提示，“Customers”、“Suppliers”和“Employees”。而另一方面，如果一个按钮代表的是打开Customers窗体而另一个按钮代表的是打印客户列表，那么你需要使用工具提示：“Open Customer Form”（或者是“Maintain Customers”）和“Print Customers”，来区分这两个按钮。

关于图像的一个主要事项：和用户界面设计的其他方面一样，将图像与含义相联系的时候，一致性是关键。在系统中代表某个实体的图标应当在任何引用该实体的地方都保持一致，同样，每个图标与一个动词之间的联系也必须一致。

最好的方法是为每一类别选择一个图像的范围——实体和动词——并且在必要的时候组合

这两类图标。例如，如果我要将X图像与删除动作联系，并且将一个人的轮廓图像与Customers表联系起来，那么我会将一个图像叠加到另一个图像上，以此表明“Delete Customer”，如图21-3所示。



图21-3 组合图像会十分富于表达力

21.2.3 状态栏

状态栏是一种为用户提供被动帮助的常见方法。它显示在窗口的底部，能够显示状态信息，比如Num Lock key、Caps Lock key以及扩展的选择模式的状态。状态栏还可以为用户显示消息。图21-4显示了一个Access状态栏。请注意，Categories窗体被最大化了，因此窗口底部的状态栏好像成为了窗体的一部分。但是，状态栏实际上是Access窗口本身的一部分。

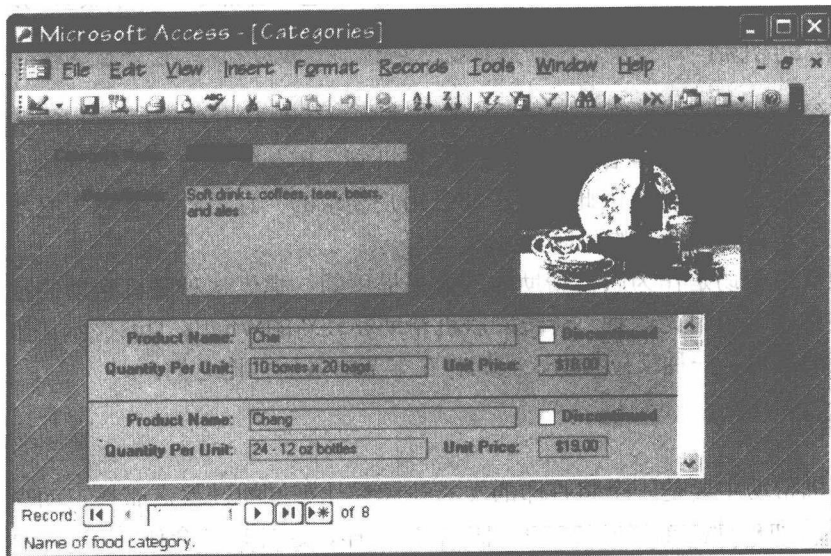


图21-4 在主窗口底部的状态栏

只要设置控件的StatusBarText属性，Access就很容易能够让你在状态栏中为窗体上的每个控件显示描述信息。如果你没有为绑定到某个字段的控件显式地定义StatusBarText属性，那么该字段的Description值（它在表的Design视图中指定）将会显示在状态栏中。但是，据我所知，在Access中不能通过代码设置状态栏，因此没有办法使用状态栏文本来为菜单项的扩展描述以及给用户的一般消息显示信息。

在Visual Basic中，可以通过设置StatusBar控件相应面板的Text属性来显式的设置在状态栏上显示的文本。因此，你能够在整个系统中使用状态栏显示解释性的信息，而不需要像Access那样只能描述活动的控件。这种额外的灵活性的代价是你必须在代码中设置文本内容，并且没有办法将文本绑定到某个控件上并让Visual Basic自动显示它。必须承认，这项额外的工作会很繁重，但是相对扩充的功能，这是十分值得的。

状态栏的最大优点是它从不强加于人。与消息框不同，状态栏不会获取键盘输入或者要求显式的用户撤令。事实上，在Access中（如果你提供该功能，那么Visual Basic中也是）用户可以选择不显示状态栏。

在Visual Basic中，你可以使用状态栏来显示扩展的描述信息或者让它们通报在后台运行

的过程的信息。你也可以使用状态栏显示错误消息，尽管你必须允许用户让状态栏有不可见的可能。使用状态栏来显示用户必须看到内容是很危险的。即使状态栏在主窗口中是可见的，用户也有可能没有注意到显示在那里的消息。

但是，在状态栏中显示消息也有其用处。正如我在第19章中推荐的，如果你暂缓而不是拒绝那些违反完整性约束的记录，那么状态栏就是一个解释这种情况以及如何解决它的很好的地方。我有时候将“问题”字段的内容显示为红色，并且当用户选择该字段的时候，就在状态栏中描述该问题并建议如何解决它。但是由于状态栏的空间有限，我也使用对话框给出更详细的说明，通常是在用户按下一个功能键之后显示它。

当然，Visual Studio.NET支持ErrorProvider控件，它以稍微不同的方式来提供相同的功能（高亮显示有问题的控件并显示一个错误消息）。当然，它的优点是不要太多的编码。

21.3 反应帮助机制

与我们已经讨论的被动机制不同，反应帮助机制只作为对用户的某些动作的响应而显示。中级用户或者高级用户比初学者更喜欢使用反应机制，因为初学者可能不知道如何调用它们或者甚至不知道它们的存在。因此，对于初级用户要求的“这是什么意思”之类的问题，反应帮助机制并不是一种特别好的帮助方法。

大多数的反应帮助是作为联机帮助的形式来提供的。提供这种联机帮助的范例已经有若干种，我们将在本节中讨论两种：传统的联机帮助和较新的What's This提示。虽然不被普遍认可，但错误消息也是一种反应帮助，我们将在本节的最后讨论它们。

21.3.1 联机帮助

传统的联机帮助主要是一个将打印的文档转译到计算机的过程，而且正如当一个现实世界的对象或者活动被计算机化时通常发生的情况一样，这种转译会使得一些事情变得容易，而同时使一些事情变得更困难。

联机帮助比打印的文档更易于使用，并且通过鼠标点击来显示交叉引用资料的功能也相当有益。而另一方面，联机帮助不便于浏览，你不能在你去喝咖啡的时候将它带到餐厅去。

细心设计帮助系统能够在某种程度上改良联机帮助的缺陷。（然而，它仍旧只能达到计算机一样的便携程度。）设计和编写帮助系统是一个很大的题目，其中的大多数内容已经超出了本书的范畴。我在这里能做的就是给你一些概要的指导，指出为数据库系统编写帮助的一些特征，并推荐一些参考书目帮你查找更详细的信息。

在为你的系统设计联机帮助的时候，首要的并且是最重要的考虑事项就是不论该帮助与系统的联系有多紧密，你都不应该将它作为用户界面的一个组成部分。这就是说，你的系统必须能够独立的存在，而不要强迫用户求助于联机帮助（或者是其他的什么文档）来完成任务。

请记住，初级用户可能不会理解联机帮助所包含的内容，因此他们很可能在迷惑的时候不会想到按F1键。你应该将联机帮助作为由系统本身提供的对用户帮助的一种支持，而不是替代它。一些设计者通常会认为联机帮助系统可以减轻他们让系统清晰明了的负担。以我的经验来看，这是错误的，并且会导致粗糙的、不可用的软件。

在设计联机帮助时需要考虑的第二件事情是你想要它提供的支持的类型。联机帮助主题可以大致分为两类：面向任务的和面向功能的。面向任务主题告诉用户如何完成一个特定的

任务，比如如何打印一张发票或者安排一场会议。面向功能主题提供了关于某项功能（比如Print命令在File菜单上）或者控件（比如CustomerID文本框在一个窗体上）的详细信息。这两类分别对应打印文档中的用户指南（User Guides）和参考手册（Reference Manuals）。

面向任务和面向功能的帮助在支持数据库系统上都各自有一定的作用。如果你的系统能够支持多个工作过程，或者它支持的工作过程十分复杂，那么面向任务的帮助可能对用户十分有用，它提供了指引过程的某种路标。不过，在系统本身中提供支持和向导仍然十分重要。仅仅给用户展示一个窗体的字母列表（比如Access数据库窗口中的那样）并且依赖联机帮助来告诉他们这些窗体必须按怎样的顺序完成是绝对不能接受的。（不管怎样，如果你是这样为我工作的，我同样也不能接受。）

理想状况下，面向任务的主题不应当包含概念或者介绍性的资料。记住，这不是一个教授用户系统能做什么的工具。面向任务的帮助主题的唯一目的就是提供一个对过程“如何”进行的摘要，不是解释“是什么”以及“为什么”。

对复杂的主题使用多个帮助标题可以保持这些标题在计算机屏幕上的可读性。如果这个被解释的工作过程很复杂并且包含多个方面，那么最好不要尝试在一个标题中解释所有方面的内容。只需要在主标题中解释最简单或者最常用的方面，并且提供到其他有不同解释的标题的链接。

与面向任务主题注重“如何”使用你的系统不同，面向功能的主题注重的是“是什么”和“为什么”。对于数据库系统，大多数面向功能的主题会涉及数据项和控件，而不是功能本身。极少数数据库系统会要求类似Access帮助所提供的主题，例如，它必须解释Mid\$函数的精确语法。

系统其实只需要那些关于解释系统中每个实体和属性以及属于它们的约束的含义的主题，它们可能还需要解释系统中不同类型的控件的使用方法——比如，如何通过树形视图控件导航，或者如何使用一个日历控件选择日期。

在计划这些面向数据的主题的时候，最重要的是考虑为什么用户要寻求帮助。如果一位用户正在关注Orders窗体上一个标题为“Desired Delivery Date”的文本框，那么他不太可能因为没有理解这是一个客户想要发货的日期而按下F1键。如果这是你的主题要告诉用户的全部内容，那么它将比一无是处还要糟糕——它惹人生气，并且浪费时间。

那么，为什么用户要按F1呢？可能他没有理解为什么这个日期已经填入了——这样就应当解释默认值以及如何覆盖它，或者可能该客户已经告诉他将货物在“月初之后的任何一个日子时候”发出——那么就应当解释他应当输入的最早日期或者任何应用于你的环境中的规则。你对用户可能要查询的问题考虑得越仔细，你的帮助系统就越有效。

21.3.2 “What's This?” 提示

除了它们被调用的方式不同之外，“What's This?” 提示更像面向任务的联机帮助。“What's This?” 提示是通过点击窗口的标题栏中的问号图标，然后点击窗口中的某个控件来调用的。图21-5显示了Access 2000中Option对话框的Windows In Taskbar 复选框的“What's This?” 提示。

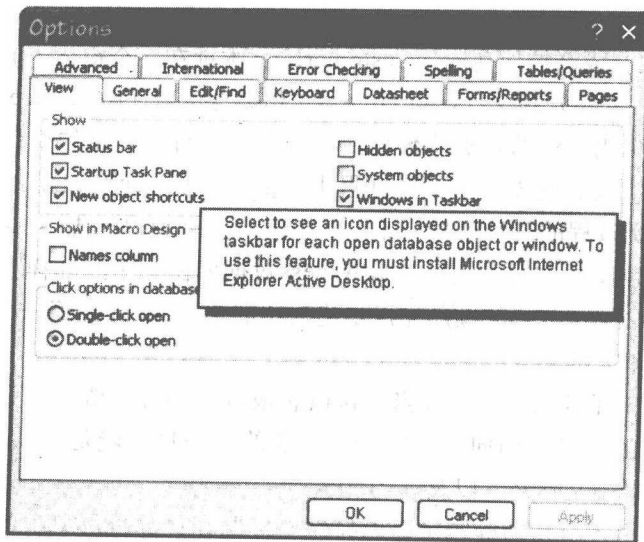


图21-5 通过点击标题栏中的问号和某个控件，“What’s This?”提示将会显示在对话框中

我喜欢“What’s This?”提示的思想，主要是因为它们与系统的用户界面紧密的集成在一起。我看到过新手在一些偶然情况下会对“What’s This?”比较迟疑。但我从未发现新手会偶然通过按F1键查询联机帮助，并且我想他们这么做可能是有一点被吓住了，这满屏幕的陌生窗口是干什么的呢？

对于一个小型系统来说，一套好的“What’s This?”主题可能就是你需要提供的所有帮助，特别是当系统不支持太多复杂工作过程的时候。然而，由于它们显示在相关联的窗口上并且不能滚动，因此“What’s This?”主题必须相当精简。如果你需要解释复杂的约束，而又没有太多空间的话，你需要使用一个或者多个较长的联机帮助主题来支援“What’s This?”主题。（切记要在“What’s This?”主题最后添加“按F1获得更详细的信息”之类的语句。）

“What’s This?”主题试图回答的特定问题就是——正如你已经从其名字猜到的一样——“这东西是什么”，可以将“What’s This?”提示看成是一个长段落的控件标签。由于空间的限制，你一般不能过于创造性地回答你认为用户可能问到的问题。但是只要多一点考虑，就一定会比仅仅提供一个重述的控件标签做得好。

将“Desired Delivery Date”描述为“客户想要发货的日期”对于“What’s This?”提示来说是十分糟糕的，在联机帮助中也一样。至少，你可以这么说：“发货的最早日期。这个值默认是在订购日期三天之后，但是你可以通过点击该字段并输入一个新的日期来修改这个值。按F1获取更详细的信息。”

21.3.3 可听见的反馈

可听见的反馈——使用计算机的语调来描述某些系统状态——是一个十分强大的帮助用户机制，但是设计者可能会很好地利用它的能力，也可能利用得很糟糕。当系统发现一个“用户错误”的时候，显示的“嘟嘟叫”的对话框就是一个糟糕使用可听见的反馈的主要例子。用户并不乐意关注他们的错误，而嘟嘟声不但将用户的注意力拽向了问题，并且它还让任何在可听范围之内的人都注意到。此外，默认的嘟嘟声确实是一种恼人的声音。

但是，如果将可听见的反馈恰到好处地使用，它将是一个十分有益的工具。与其让系统在迷惑的时候对用户发出嘲笑声，不如在事情正确的时候让计算机发出柔和的声音（我认为它是咕噜声）。比如，当数据输入时，系统在用户离开相应控件的时候检查每个字段。如果数据符合所有相关约束，那么系统就发出一个柔和的“I'm OK”的声音。（这个声音不应当太大。）如果发生了任何问题，那么系统不发声，而只是在状态栏上显示一个消息。静音就是对问题足够的提醒，并且用户会查看屏幕的。

与这种积极的反馈技术类似的是键盘。当你按下每个键时，键盘就发出一个柔和的按键声。你可能没有注意到这个声音，但是如果它停止了，你就会立刻注意到并且开始检查问题。

键盘的例子应当能平复你对一个满是数据录入人员的房间里可能存在的杂音的恐惧。正如我已经说过的，“I'm OK”的声音不需要过分突出。我已经为一个呼叫中心的系统成功使用了这种积极的可听见的反馈技术，该中心在一个房间里有超多100名的用户。

21.3.4 错误消息

非常遗憾大多数人不认为错误消息是一种为用户提供帮助的方式。更遗憾的是有些人不但认为大多数错误消息都对用户没有帮助，反而是对用户的责难。不要将错误消息看作是指出用户错误的机会，而应当将它们作为用户帮助的一种请求。系统会陷入麻烦中，它需要用户帮助来排除困难。

一位行为端正的人在请求他人帮助的时候不会作隐晦的陈词或者发布命令；一位行为端正的人不会在她迷惑或者遇到麻烦的时候试图暗示这是其他人的错误；一位行为端正的人会尽可能清楚地解释问题，礼貌地请求帮助，不会强加任何过多的要求，并且会尽力解释她请求的含义。

一个运作良好的计算机系统也应当如此。事实上，因为计算机系统赢得的尊重比人要少，因此它们应当做得更多。（卑躬屈膝在这里不合适。）当发布“错误”消息的时候，系统有责任做到以下几点：

- 清楚地解释状况，并使用用户理解的术语。
- 礼貌地请求帮助，不要暗示用户错了。
- 不要强制要求用户作任何系统自身能够做到的事情。
- 描述用户可能做的任何动作的含义。

系统有时候会让自己陷入迷惑，或者在某些环境因素中，比如内存短缺或者磁盘错误，如果没有用户的帮助，这些错误常常会让系统无法继续运作。当发生这些事件的时候，你别无选择地只有给用户显示一条消息。这不仅是让系统得到需要的帮助的绝好机会，也是提供给用户帮助系统摆脱麻烦的信息。

通过尽可能清楚地解释情况并且不使用让人迷惑的术语，你显示给用户的消息将让他们以正确的方式解决问题。在首先理解问题的基础上，用户会避免在将来遇到同样的问题——假定该情形是可以避免的。无效的数据输入格式是可以避免的；磁盘错误通常不可能避免。

通过解释用户选择的含义，系统更易接受容易察觉的响应。请记住很多看上去对你十分显然的事情对用户来讲就不那么明显了。当然，不要下指令，但是也不要担心阐述明显的事情。

礼貌地描述消息并且不强制给用户是个很好的习惯。还记得你的母亲告诉过你，使用蜂蜜会比醋抓到更多的昆虫吗？如果用户发觉你的系统十分礼貌和有益，那么他们会倾向原谅

你没能追捕到那个奇异的GPF (General Protective Fault, 一般保护错)。

21.4 主动帮助机制

被动和反应用户帮助都是相对易于理解的用户支持类型, 虽然它们随着我们对人机交互知识的增长以及新的实现工具的应用也变得越来越复杂和成熟。最后一种类型——主动帮助, 更像是这个领域的新生儿, 并且目前极少有系统实现了它。

主动用户帮助的原则很简单: 计算机系统监视用户的行为并且主动地进行帮助, 或者是给出更有效地使用系统的建议, 或是代表用户执行任务。Microsoft Office Assistant通过在用户行为的基础上给用户提示的方式提供了主动帮助。

当前备受关注的一种主动帮助就是智能助手。一个智能助手就是一个用户能够将任务委托给它的软件。智能助手通常会作为一个专门的网页用户界面实现, 它可以回答诸如“查找该项目最佳的价格”或者“给我推荐一本我喜欢阅读的书”之类的问题。但是并不一定要将它们限制在网页环境下。例如, 你可以以学生喜欢的方式实现一个帮助他们安排课程计划的智能助手——“在中午之前不要安排任何数学课, 并且我喜欢在晚上学语言课程”。

Microsoft提供了两个基于卡通形象来创建界面的工具, Microsoft Office Assistant和Microsoft Agent。大多数人对用一个弹跳的回形针来表示Microsoft Office Assistant (用户可以改变形象) 十分熟悉, 但是不是所有的人都能认识到Office Assistant有一个可编程的界面。不过, Office Assistant仅仅在Microsoft Office应用程序内部有效, 并且不能通过Access运行时引擎发布。

如果你是在Access中工作 (或者任何其他可以使用Microsoft ActiveX控件的开发环境, 包括Visual Basic), 那么你可以从Microsoft网站上下载Microsoft Agent SDK。它比Office Assistant更加强大。

Microsoft Agent是一个十分酷的工具。除了SDK中提供的之外, 你可以设计自己的动画形象。Microsoft Agent还支持语音识别, 这对于数据库应用程序来说是一个巨大的潜力。我很想看到Microsoft Agent 界面出现在Microsoft SQL Server 中的Microsoft English Query (提供自然语言处理的SQL查询的功能) 工具中。想象一下, 将数据库本身描述为一个Microsoft Agent形象。

但是, 需要警告的是, Office Assistant和Microsoft Agent都不能为你监视和响应用户动作提供直接支持。它们提供了简单的界面来丰富与用户之间的文字交流, 但是当它真正实现智能化的时候, 你就得完全独自完成了。

21.5 用户培训

对于用户培训的要求, 数据库系统和其他任何类型的软件都一样。最初的用户培训可以通过文档、教室讲座或者计算机来提供。并且这些选择都不互相排斥。

如果你决定实现一个基于计算机的培训机制, 那么要清楚项目的范围和听众。初级用户主要对系统能做什么感兴趣, 只有高一级的用户才对如何实现感兴趣。

针对初级用户的培训的范围根据系统的复杂程度以及预算的不同而有所区别。很多系统只要求一个介绍性的界面或者对系统的一些解释。较复杂的系统将会从更宽泛的培训机制中受益, 这可能是一个指导性的漫游, 或者甚至是正式的含有练习和测试的计算机化的培训。

中级用户主要对如何实现特定任务感兴趣，并且很多情况下，面向任务的帮助就能满足他们的要求。事实上，“帮助”和“培训”的区别在某种程度上是随意的。但是，复杂的系统对中级用户的培训可能和初级用户一样要求正式的计算机化的培训。

不论你实现的培训的范围如何，一定要保证培训与系统本身分离开来。用户应当能够从系统的用户界面上，或者可能是从帮助菜单的某一项开始培训。但是培训资料的存在不应当与正式的系统使用相互干扰。

21.6 小结

在本章，我们讨论了三种用户帮助的类型：嵌在系统用户界面中的被动帮助，作为某些用户动作结果而发生的反应帮助，以及由系统本身发起的主动帮助。

我们在开始部分讨论了被动帮助的三种类型：帮助记忆的访问键、工具提示以及状态栏。之后我们讨论了反应帮助，它通常是以联机帮助或者What's This? 提示的形式实现的，但是也包括可听见的反馈和错误消息。最后，我们简要探讨了主动帮助的新领域以及用户培训。

术 语 表

- abstract entity (抽象实体):** 对实体间的联系进行建模的实体。
- ad hoc report (特殊报表):** 在应用程序完成后由用户配置的报表。
- aggregate function (聚集函数):** 一种返回统计值的SQL函数。
- alternate key (替换键):** 关系中的不作为表的主码使用的一个候选码。
- application (应用程序):** 用户可以进行交互的窗体和报表。
- attribute (属性):** 关系中的一个列。
- base relation (基本关系):** 在数据库中被实例化为一个表的关系。
- binary relationship (二元联系):** 有两个参与者的联系。
- Boolean expression (布尔表达式):** 结果为真或假的表达式。
- business constraint (业务约束):** 来自于问题域的约束。
- business rule (业务规则):** 来自于问题域而不是关系理论的完整性约束。
- candidate key (候选码):** 唯一标识一个关系的一个或多个属性。
- cardinality of a relation (关系的基数):** 关系中行的数量。
- cardinality of a relationship (联系的基数):** 可以参与到一个联系中的一个实体的最大实例数。
- Cartesian product (笛卡儿积):** 将一个结果集中的每个记录和另一个结果集中的每个记录组合起来的关系运算。
- cascading delete (级联删除):** 当删除主表中的记录时自动删除外码表中的相应记录。
- cascading update (级联更新):** 当更改主表中的记录时自动修改外码表中的相应记录。
- closure (闭包):** 在关系上进行的任何操作的结果都是可进一步用于其他操作的关系原则。
- command vector (命令向量):** 在用户界面上一个命令的执行路径, 例如, 一个菜单项或者一个工具栏按钮。
- composite entity (复合实体):** 被一个或多个关系建模的问题域中的一个实体。
- composite key (复合键):** 由两个或多个属性构成的候选码。
- concrete entity (具体实体):** 对现实世界中的对象或事件建模的一个实体。
- conventional value (常值):** 用于表达一个不存在的或者未知的值的一个任意值。
- database (数据库):** 数据库模式以及存储的数据的组合。
- database access object model (数据库访问对象模型):** 用于在数据库引擎和开发环境中进行通信的一个软件库。
- database application (数据库应用程序):** 用户可以进行交互的窗体和报表。
- database constraint (数据库约束):** 引用多个关系的一个完整性约束。
- database engine (数据库引擎):** 处理数据的物理存储和操作的软件。
- database schema (数据库模式):** 数据库中表的物理设计。
- database system (数据库系统):** 数据库应用程序、数据库引擎和数据库的组合。

data integrity (数据完整性): 数据库使用的用于确保数据至少是可用的 (如果不正确的) 规则。

data model (数据模型): 关系术语中的问题域的概念描述。

declarative integrity (声明完整性): 定义完整性约束的一种方法, 通过作为表定义的一部分来显式地声明它们。

degree of a relation (关系的度): 一个关系中列的个数。

degree of a relationship (联系的度): 一个联系中参与者的数量。

derived relation (派生关系): 依据其他关系定义的一个虚拟关系。

domain (域): 属性可以取值的范围。

domain constraint (域约束): 决定域的可能的取值范围的完整性约束。

entity (实体): 关于系统需要存储的信息的任何事物。

entity constraint (实体约束): 确保被系统建模的实体的有效性的完整性约束。

equi-join (等值连接): 两个表间的基于相等的连接。

field (字段): 数据库中的属性的物理表示。

foreign relation (外码关系): 接收联系中的其他参与者的主码的关系。

full outer join (全外连接): 返回两个参与者中的全部字段的一个外连接。

functional dependency (函数依赖): 两个属性之间的联系, 使得一个属性中的值决定另一个属性中的值。

inner join (内连接): 一种连接, 只返回操作结果为真的记录。

integrity constraint (完整性约束): 一个数据完整性规则。

intrinsic constraint (内置约束): 管理数据库的物理结构的约束。

join dependency (连接依赖): 三个关系间的循环联系。

junction table (连系表): 表达数据库中的联系的表。

left outer join (左外连接): 返回SELECT语句中第一个记录集中的全部字段的外连接。

logical data type (逻辑数据类型): 域的一般表示, 比如“字符串”或“数字”, 不引用特定的物理数据类型。

logical operator (逻辑操作符): 返回布尔结果的运算符。

lossless decomposition (无损分解): 使分解的关系可以被重新组合起来而不会丢失任何信息的能力。

multivalued dependency pairs (多值依赖对): 参与在函数依赖中的相互独立的属性集。

natural join (自然连接): 等值依赖的一种特例, 这种连接的实现是基于相等, 所有的公共列都参与到连接中, 但在结果集中只包含一个公共字段集。

normal forms (范式): 定义关系的结构的规则, 以便消除更新异常。

normalization (规范化): 结构化数据库模式的过程, 以确保数据完整性, 消除冗余以及便于数据检索。

orphan entity (孤立实体): 在一个联系中与主关系中的实体不关联的弱实体。

outer join (外连接): 返回内连接中的所有记录加上其他参与者中的一个或全部记录。

partial participation (部分参与): 在一个特定联系中可以独立于它的参与者而存在的实体。

participant (参与者): 与联系中的其他实体关联的一个实体。

passive user assistance (被动的用户支持): 是用户界面内在部分的一个用户辅助机制。

primary key (主码): 一个关系中用于唯一标识表中的一条记录的候选码。

primary relation (主关系): 主码被存储在联系中的其他参与者中的关系。

proactive user assistance (预见性用户支持): 试图预见用户需求的一个用户辅助机制。

problem space (问题域): 被数据库应用程序建模的现实世界的一部分。

procedural integrity (过程完整性): 通过创建在记录被修改、插入或删除时自动执行的代码来增强数据完整性的一种方法。

query (查询): 一种在Microsoft Access中派生的关系。

reactive user assistance () 响应性用户支持: 由用户的一些活动触发的一种用户辅助机制, 比如有效的登录或对在线帮助的要求。

record (记录): 元组的物理表达。

recordset (记录集): 在Microsoft Access中使用的一个普通术语, 表示一个关系的物理表达。

referential integrity (参照完整性): 确保实体间的联系保持有效的完整性约束。

regular entity (常规实体): 可以不参与在联系中而存在的实体。

relation (关系): 将数据组织成行和列的逻辑结构。

relation body (关系体): 组成关系的元组。

relation heading (关系头): 对关系的顶部的属性和域的定义。

relational database (关系数据库): 由Dr.E.F.Codd定义的关系模型的物理实现。

relational difference (关系差): 返回与另一个记录集中的记录不匹配的某个记录集中的记录的关系操作。

relational divide (关系除): 返回一个记录集中的全部记录的一种连接, 只要这些记录的值匹配另一个记录集中的全部相应记录。

relational intersection (关系交集): 返回两个记录集中的全部相同记录的关系操作。

relational union (关系合并): 两个记录集的串联。

relationship (联系): 两个或多个实体间的一个关联。

right outer join (右外连接): 返回SELECT语句中列在后边的记录集中的全部字段的外连接。

scalar value (标量值): 一个单一的、不重复的值。

schema (模式): 数据库系统中表的物理结构。

simple key (简单键): 仅由一个属性组成的候选码。

standard report (标准报表): 可以作为数据库应用程序的一部分定义和实现的报表。

table (表): 在数据库模式中关系的物理实例。

task (任务): 工作过程中的一个离散步骤。

ternary relationship (三元联系): 有三个参与者的联系。

theta-join (θ 连接): 从技术角度来说, 任何连接都是基于比较运算符的, 但通常指的是相等之外的其他运算符的连接。

three-valued logic (三值逻辑): 允许表达式的结果为真、假和空三个值的逻辑模型。

total participation (完全参与者): 必须参与在特定联系中的实体。

transaction integrity (事务完整性): 控制数据库中的多个操作的一种完整性约束。

trigger (触发器): 当指定的数据库事件发生时执行的过程代码。

tuple (元组): 关系中的一行。

type-compatible domains (类型兼容的域): 可以被逻辑地比较的域。

unary relationship (一元联系): 一个关系和它自己之间的关联。

update anomaly (更新异常): 由设计不好的数据模型引起的数据操作问题。

view (视图): 在SQL Server中一个派生的关系。

weak entity (弱实体): 仅当参与了一个给定的联系才存在的实体。

work process (工作过程): 将要由数据库应用程序实现的工作。

参考文献

一、关系数据库理论

Date, C. J. *An Introduction to Database Systems*. 7th Edition. Reading: Addison-Wesley Publishing Company, 1999.

Date, C. J. and Hugh Darwen. *Foundation for Object/Relational Databases: The Third Manifesto*. Reading: Addison-Wesley Publishing Company, 1998.

Fleming, Candace C. and Barbara von Halle. *Handbook of Relational Database Design*. Reading: Addison-Wesley Publishing Company, 1989.

Teorey, Toby J. *Database Modeling & Design*. 3rd Edition. San Francisco: Morgan Kaufmann Publishers, 1999.

二、关系数据库系统的设计

Gilb, Tom and Susannah Finzi. *Principles of Software Engineering Management*. Reading: Addison-Wesley Publishing Company, 1988.

Haught, Dan and Jim Ferguson. *Microsoft Jet Database Engine Programmer's Guide*. 2nd Edition. Redmond: Microsoft Press, 1997.

McConnell, Steve. *Rapid Development*. Redmond: Microsoft Press, 1996.

Pressman, Roger S. *Software Engineering: A Practitioner's Approach*. 3rd Edition. New York: McGraw-Hill, 1992.

Sommerville, Ian. *Software Engineering*. 6th Edition. Reading: Addison-Wesley Publishing Company, 1996.

Soukup, Ron. *Inside Microsoft SQL Server 6.5*. Redmond: Microsoft Press, 1997.

三、用户界面的设计

Cooper, Alan. *About Face: The Essentials of User Interface Design*. Foster City: IDG Books Worldwide, 1995.

Heckel, Paul. *The Elements of Friendly Software Design*. New York: Warner Books, 1991.

Mandel, Paul. *The Elements of User Interface Design*. New York: John Wiley & Sons, 1997.

Microsoft Corporation. *The Windows Interface Guidelines for Software Design*. Redmond: Microsoft Press, 1998.

Shneiderman, Ben. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Reading: Addison-Wesley Publishing Company, 1980.